Efficient Dependability Assessment of Systems Software

TECHNISCHE UNIVERSITÄT DARMSTADT

Vom Fachbereich Informatik
der Technischen Universität Darmstadt

genehmigte

Dissertation

zur Erlangung des akademischen Grades eines
Doktor-Ingenieur (Dr.-Ing.)

vorgelegt von

Nicolas Coppik, M.Sc.

aus Frankfurt am Main

Referenten:
Prof. Dr. Matthias Hollick
Prof. Neeraj Suri, Ph.D.
Prof. Dr. Stefan Katzenbeisser

Tag der Einreichung: 15. April 2020
Tag der mündlichen Prüfung: 28. Mai 2020

Darmstadt, 2020
D17

Nicolas Coppik: *Efficient Dependability Assessment of Systems Software*
Darmstadt, Technische Universität Darmstadt
Tag der mündlichen Prüfung: 28.05.2020

Jahr der Veröffentlichung der Dissertation auf TUprints: 2020
URN: urn:nbn:de:tuda-tuprints-118290
URL: https://tuprints.ulb.tu-darmstadt.de/id/eprint/11829

Efficient Dependability Assessment of Systems Software

by

Nicolas Coppik

Erklärung

Hiermit versichere ich, die vorliegende Dissertation selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Eigenzitate aus vorausgehenden wissenschaftlichen Veröffentlichungen werden in Anlehnung an die Hinweise des Promotionsausschusses Fachbereich Informatik zum Thema „Eigenzitate in wissenschaftlichen Arbeiten“ (EZ-2014/10) in Kapitel „Collaborations and Contributions“ auf Seiten xi bis xii gelistet. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, 15. April 2020

_____

Nicolas Coppik

Abstract

Computing systems and the various services and applications they enable have become pervasive in our daily lives. We increasingly rely on these complex systems, including many systems built on general purpose hardware and software, to consistently provide important functionality. As we grow more and more dependent on such systems, we need to ensure that they are, in fact, dependable and that we can trust their ability to consistently provide the functionality we expect from them. Therefore, we need techniques for assessing and improving the dependability of such systems. To be practical, such techniques must not only be applicable to complex software systems, they need to scale with their increasing sizes.

Common approaches to improve the dependability of software systems include testing techniques to find faults and dependability issues as well as techniques intended to predict the impact of residual software faults. Software Fault Injection (SFI) is an approach that can be useful in both contexts, for finding dependability shortcomings and estimating the impact of residual faults, whereas most other testing techniques, such as fuzzing, are primarily used to find faults. Many approaches to improve software dependability suffer from scalability issues and are difficult to apply to large, complex software systems, and particularly to systems software, such as operating system kernels.

With this general background in mind, this thesis aims to improve the efficiency and precision of SFI techniques for systems software, as well as to develop novel guidance mechanisms for feedback-driven fuzzing.

We develop a technique to trace error propagation in monolithic operating system kernels, apply it to modules from the widely used Linux kernel, and show that conventional oracles for SFI tests can misclassify a substantial fraction of seemingly successful executions. We then focus on accelerating SFI experiments since, due to increasing software complexity, comprehensive SFI testing requires an increasing amount of test executions, which in turn leads to long test latencies. Starting with user mode software, we develop a novel execution model that uses static and dynamic analysis to avoid redundant code re-execution and facilitates parallelization. Since long SFI test latencies are particularly problematic for systems which may require additional instrumentation to trace error propagation, we then develop a related approach to accelerate SFI experiments for kernel code, and apply it to the Linux kernel using error propagation analysis instrumentation and achieve substantial speedups. Finally, we develop a novel guidance mechanism for feedback-driven fuzzing that makes use of input-dependent memory accesses in the target program.

Zusammenfassung

Die vielfältigen Dienste und Anwendungen, die durch moderne Computersysteme ermöglicht werden, sind in unserem Alltag allgegenwärtig. Wir verlassen uns zunehmend auf komplexe Computersysteme um wichtige alltägliche Funktionen bereitzustellen. Dabei handelt es sich oft um Systeme, deren Soft- und Hardware nicht spezialisiert, sondern aus Standardkomponenten konstruiert ist. Da wir uns mehr und mehr auf solche Systeme verlassen, müssen wir sicherstellen, dass sie zuverlässig sind und die von ihnen erwartete Funktionalität durchgehend bereitstellen können. Folglich benötigen wir Techniken, die geeignet sind, die Zuverlässigkeit solcher Systeme zu überprüfen und zu verbessern. Praxistaugliche Techniken müssen sowohl auf komplexe Softwaresysteme anwendbar sein als auch mit deren zunehmender Größe skalieren.

Verbreitete Ansätze zur Verbesserung der Zuverlässigkeit von Softwaresystemen sind Testverfahren, mit denen Fehler und Zuverlässigkeitsmängel gefunden werden können, sowie Verfahren zur Vorhersage der möglichen Auswirkungen im System verbleibender Fehler. Softwarefehlerinjektionstechniken (SFI-Techniken) können in beiden Kontexten genutzt werden, sowohl um Zuverlässigkeitsmängel zu finden als auch um die Auswirkungen verbleibender Fehler abzuschätzen. Andere Techniken, wie Fuzzing, dienen in erster Linie dazu, Fehler zu finden. Viele Verfahren zur Verbesserung der Softwarezuverlässigkeit leiden unter Skalierbarkeitsproblemen und sind schwer auf große, komplexe Softwaresysteme, insbesondere auf Systemsoftware wie den Betriebssystemkernel, anwendbar.

Vor diesem Hintergrund zielt diese Dissertation darauf ab, die Effizienz und Präzision von SFI-Techniken für Systemsoftware zu verbessern sowie neue Steuerungsmechanismen für feedbackbasierte Fuzzing-Verfahren zu entwickeln.

Wir entwicklen ein Verfahren zur Nachverfolgung der Ausbreitung von Fehlerauswirkungen in monolithischen Betriebssystemen, wenden es auf Module aus dem weit verbreiteten Linux-Kernel an und zeigen, dass konventionelle Verfahren einen beträchtlichen Anteil der anscheinend erfolgreichen SFI-Testausführungen falsch klassifizieren. Da mit zunehmender Softwarekomplexität gerade auch SFI-Techniken unter Skalierbarkeitsproblemen leiden, wenden wir uns dann der Beschleunigung von SFI-Tests zu. Zunächst entwickeln wir ein neuartiges Ausführungsmodell für im Benutzer-Modus laufende Software, welches, unter Verwendung von statischer und dynamischer Analyse, die wiederholte, redundante Ausführung von Programmcode vermeidet und Parallelisierung begünstigt. Da lange SFI-Tests für Sys-

teme, welche zusätzliche Instrumentierung zur Nachverfolgung der Auswirkungen von Fehlern benötigen, ein besonders großes Problem sind, entwickeln wir einen Ansatz, der auch SFI-Tests für Betriebssysteme beschleunigen kann. Wir wenden diesen Ansatz auf instrumentierten Code aus dem Linux-Kernel an und erreichen eine erhebliche Verkürzung der SFI-Testdauer. Zuletzt entwickeln wir einen neuartigen Mechanismus zur Steuerung von feedbackbasierten Fuzzing-Verfahren, welcher eingabeabhängige Speicherzugriffe im getesteten Programm nutzt.

Collaborations and Contributions

The results and contributions presented in this thesis are, like the corresponding publications, the result of collaborative efforts by several authors. Clearly delineating individual contributions is challenging as each author contributed to discussions of ideas, prototype implementations, experimental analysis, or the presentation of results to varying degrees. In the following, I briefly outline which publication each chapter of this thesis is based on and what the contributions of my co-authors and myself are.

Chapter 1 provides the introduction to this thesis, including background information and an overview of the research questions, contributions, and publications. It is not based on previously published material.

Chapter 2 is based, in parts verbatim, on material from [Cop+17], which is joint work with Oliver Schwahn, Stefan Winter, and Neeraj Suri. I developed the approach based on discussions with my co-authors, implemented the TrEKer prototype, and conducted the evaluation. Additional scripts used in the evaluation were contributed by Oliver Schwahn. The analysis of the experimental results was performed jointly by Oliver Schwahn, Stefan Winter, and myself.

Chapter 3 is based on material previously published in [Sch+18a], which is joint work with Oliver Schwahn, Stefan Winter, and Neeraj Suri. The FastFI approach was developed by Oliver Schwahn and myself following discussions about the applicability of a fork server architecture to software fault injection experiments. I developed the initial prototype of the version library generation and contributed to the development of the static analysis and runtime logic components. The experimental evaluation and analysis were joint work by Oliver Schwahn, Stefan Winter, and myself.

Chapter 4 is based, in parts verbatim, on material which is currently under submission [CSS20]. It is joint work with Oliver Schwahn and Neeraj Suri. I developed the enhanced execution model and the integration of the TrEKer tracing instrumentation, implemented the controller including all host-side runtime logic, and adapted the TrEKer error propagation analysis. I also contributed to the runtime kernel module used in the guest virtual machines, which was developed primarily by Oliver Schwahn, who also contributed most of the workload scripts and performed initial testing on the evaluation targets. I conducted the experimental evaluation. The analysis of the results was performed jointly by Oliver Schwahn and myself.

Chapter 5 is based, in parts verbatim, on material from [CSS19], which is joint work with Oliver Schwahn and Neeraj Suri. I developed the approach of using input-dependent memory accesses to guide a feedback-driven fuzzer. The implementation was joint work by myself and Raik Joachim. I conducted all experiments for the evaluation and the analysis of the results. Oliver Schwahn contributed to the analysis of the results.

Chapter 6 contains concluding remarks and is not based on previously published work.

Contents

1 Introduction

Computing systems have become pervasive in our daily lives, including everything from low-power embedded systems in home electronics or automotive applications, small Internet of Things (IoT) devices, increasingly capable hand held devices such as modern smartphones, and powerful server systems. As these systems have become more and more widespread, they have also become increasingly complex, and we have started to rely on them to consistently provide important functionality. For instance, smart home devices that require computing systems to adjust a thermostat, unlock a door, or turn on the lights are becoming increasingly common. While the importance of dependability and reliability in more conventional embedded applications, such as in the automotive and aerospace sectors, is well understood, devices in other application domains — e.g., smart home devices — commonly operate with off-the-shelf software stacks and run general purpose operating systems such as Linux. For instance, Linux-based systems are used in smart lighting systems [Phi], smart speakers [Bro18], as well as in the vast majority of smartphones sold today [IDC20] and numerous other Android-based devices. Such systems frequently receive updates that add new functionality and do not always undergo rigorous testing since, unlike conventional safety critical systems, they are not subject to international safety standards such as ISO 26262 [Int11] or IEC 61508 [Int10], which impose development and quality assurance processes. This can result in shortcomings in dependability and security, which in turn can have a substantial impact on users, for instance, when software bugs cause smart thermostats to drain their battery and turn off [Kni16], security issues let attackers bypass smart locks [Lyn18], and security cameras and smoke alarms are affected by server outages [Mat16]. It is now common for many such devices to be constantly connected to the internet, rendering security and dependability even more important, as they may be exposed to untrusted inputs at any time. The Cisco Annual Internet Report (2018–2023) forecasts that there will be more than three times as many devices connected to IP networks as there are people on earth, with IoT devices, and smart home devices in particular, taking an increasing share [Cis20].

All these devices require an extensive amount of complex software in order to provide their functionality, both on the device itself and, in the case of cloud-based systems, on the backend server systems. This software is commonly organized in a software stack in which components on the upper layers, such as application software, depend on the lower layers, such as the operating system, middleware, and

system libraries. We refer to software on the lower layers of the stack as systems software. Systems software manages system resources, and provides a platform and basic functionality to other software on the upper levels of the software stack, such as application software. As higher level components rely on lower level components to provide services that they require to implement the desired functionality, it is important for the latter to operate dependably. If low level components fail to operate correctly, the system as a whole can be impacted, and higher level components may in turn fail to provide the intended functionality, either because the system crashes or becomes non-responsive, or due to more subtle effects such as state corruption, which can be difficult to detect and debug.

The question then arises how the dependability of systems software can be assessed and improved. There are four complementary means to achieve dependability and security [Avi+04]:

1. *Fault prevention*, which encompasses techniques intended to avoid the introduction of faults into a system in the development phase;

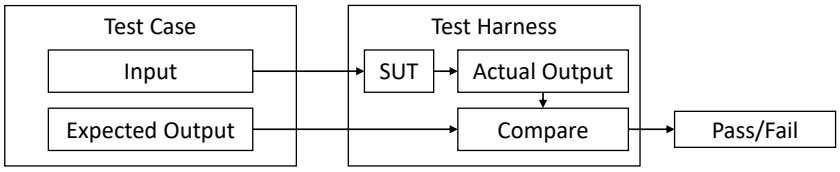
2. *Fault tolerance*, which encompasses error detection and recovery mechanisms to prevent faults from leading to failures;

3. *Fault removal*, which encompasses techniques to find and remove faults from a system; and

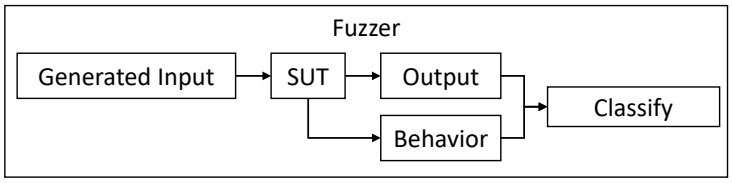
4. *Fault forecasting*, which encompasses techniques that aim to estimate the number of faults in a system, and their consequences when they are activated.

Techniques for assessing and improving the dependability and security of software systems fall into one or more of these categories. We focus on the latter two means, fault removal and fault forecasting.

Fault removal encompasses, for instance, software correctness testing, with the goal of finding and fixing as many faults as possible before deploying a system. In correctness testing, a system or component (termed the System Under Test (SUT)) is exposed to known inputs and its outputs are compared to known correct values. If the values match, the test has passed. The mechanism used to determine whether a test has passed or failed is called the test oracle. A simplified version of this process is illustrated in Figure 1.1a. Other, related approaches in the fault removal category work by subjecting a system to automatically generated inputs. Since the corresponding, correct output values for such inputs are not known, ascertaining whether the SUT has produced the correct output is difficult. This is known as the oracle problem. One way of tackling it is to check whether the system exhibits known, undesirable failure modes as opposed to comparing to known, concrete output values as is done for manually written tests. This kind of testing aims to ensure that the system behaves as desired even when faced with malformed or maliciously



(a) Correctness Testing



(b) Random Testing or Fuzzing

Figure 1.1: Correctness Testing and Random Testing

crafted inputs. By checking for specific, undesirable behaviors rather than verifying the correctness of outputs, the oracle problem can be largely avoided in this kind of testing. One approach that falls in this category and is widely used to find security and robustness issues is fuzzing. A simplified version of this approach is shown in Figure 1.1b. Fuzzing comprises a variety of random testing techniques that use different forms of input generation and are tailored to different target systems. The illustration therefore omits these details. Inputs can be generated from scratch or by mutation and different fuzzing approaches differ substantially in which aspects of the fuzzing target's behavior and output they monitor. For security and robustness testing, fuzzing is commonly used to find inputs that cause the SUT to crash or trigger memory safety violations. The fault removal category also encompasses techniques that aim to assess the fault tolerance mechanisms, such as error detection and recovery, in a system and can therefore overlap with the fault forecasting category.

Faults that are not detected during testing and exist in deployed software systems are termed residual faults, and fault forecasting techniques are intended to assess the effects of such faults on the behavior of the system. A common technique from this category is Software Fault Injection (SFI) [DM06], whereby faults in the form of code mutations are deliberately introduced to a component in the system, and the effect on other components and overall system behavior is observed. The injected faults are intended to be representative of the residual faults that are present in a system [CN13; Nat+13; Nat11]. SFI is related to but distinct from other FI techniques which focus on subjecting a system to simulated hardware faults, such as bit flips in main memory [CP95; HTI97]. While FI is well-established for testing fault tolerance mechanisms, particularly in safety critical systems, SFI techniques

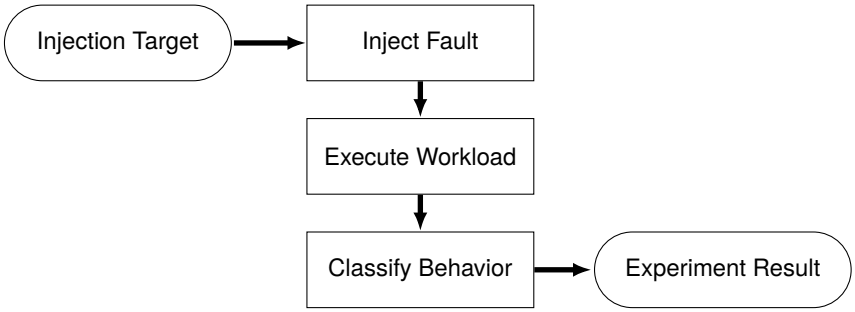


Figure 1.2: Basic Software Fault Injection Workflow

are particularly well-suited to assessing the dependability of systems that rely on commodity software components. Due to the complexity of modern software stacks and the widespread reuse of components by different vendors with varying levels of quality assurance in different operational contexts, most such systems are likely to contain residual software faults. With SFI, testers can introduce faults that resemble such residual faults and observe how the system as a whole or specific other components are affected. As it is suitable both for fault forecasting and for assessing a system's fault tolerance mechanisms, SFI falls into both the fault removal and fault forecasting categories.

In common SFI techniques, a fault is introduced into a target component in the system, a predefined workload is executed, and the behavior of the system is checked for divergences compared to a fault-free execution, typically by checking for differences in the resulting output and undesirable behaviors, such as crashes, hangs, or error indicators. This basic workflow is shown in Figure 1.2. This workflow is repeated for each fault that the system is tested with, which can result in thousands or tens of thousands of test executions. Therefore, to keep test latencies manageable, it is desirable to keep the workload short and terminate the experiment as soon as the workload has finished executing. Unfortunately, this limits the utility of such an approach for a number of important application domains. For instance, embedded systems, IoT devices, automotive systems, or operating systems are typically long-running systems, and the lack of divergences after the execution of a specific workload is not sufficient to determine whether a fault affected the system state in a manner that may affect system behavior at a later time. Addressing this requires more fine-grained monitoring of the SUT, which in turn further exacerbates the problem of long execution latencies. To mitigate this problem, techniques to accelerate SFI-based dependability assessment are needed.

We focus on fuzzing and SFI techniques as ways to detect and remove faults from a system, as well as assessing the impact of residual faults on its behavior. Unfortunately, both kinds of techniques suffer from long execution latencies, particularly when applied to large scale, complex software systems.

With this general background in mind, this thesis

1. develops an automated approach for determining how faulty components in monolithic operating systems may affect other parts of the system using memory access instrumentation, thereby facilitating the use of SFI techniques for long-running software systems;

2. develops a technique to speed up SFI experiments by avoiding redundant executions of common execution prefixes, omitting test executions for faults that cannot be triggered by a given workload, and effectively utilizing modern parallel processors;

3. develops an approach for applying such an SFI acceleration to speed up SFI experiments on instrumented kernel code;

4. develops a technique that uses memory access instrumentation to augment evolutionary fuzzing.

The shared goal of all developed approaches and techniques is to improve dependability and security assessment techniques, especially with respect to efficiency. These techniques have been developed with applicability to the lower levels of the software stack in mind and have been tested on software that can generally be categorized as belonging to these lower levels. A substantial portion of the work described in this thesis deals with kernel code, as the dependability of kernel code is crucial to the dependability of the overall system.

In the remainder of this chapter, we first give additional background on the role of systems software in the software stack in Section 1.1. We then cover dependable and secure systems and software in Section 1.2. Section 1.3 introduces the research questions and contributions of this thesis, Section 1.4 covers related publications, and Section 1.5 outlines the structure of the remainder of the thesis.

1.1 Systems Software

Software systems can be understood as collections of interacting components that are organized in a broadly hierarchical structure called the software stack. In this section, we describe this view of software systems and discuss which parts of the stack can be categorized as systems software. We also discuss the implications of dependability and security issues at different levels of the software stack on a system.

A simplified illustration of the software stack is shown in Figure 1.3. At the bottom of the stack are hardware and firmware, which are outside of the scope of this discussion. The next level up in the stack comprises the Operating System (OS) kernel. Its task is to interact with the hardware, manage the available system resources,

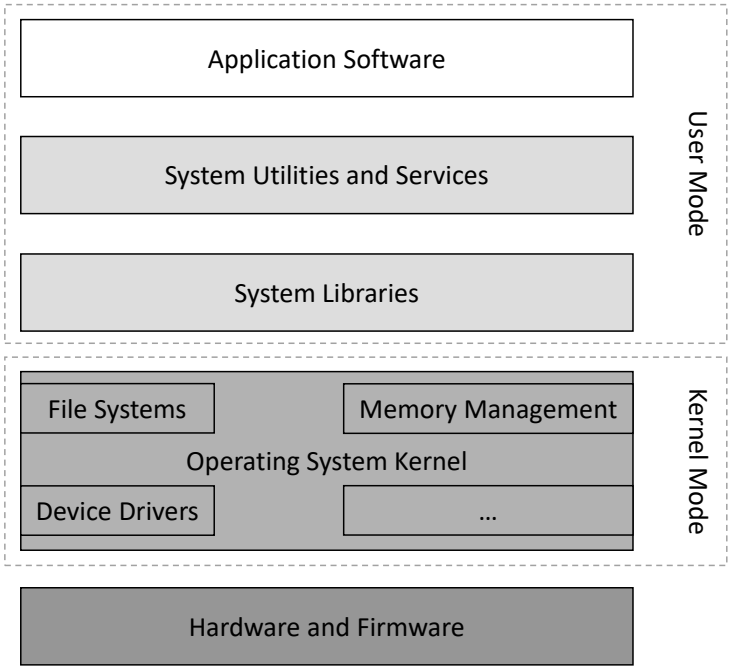


Figure 1.3: The Software Stack

and provide services and interfaces to software higher up in the stack. It also enforces isolation and security boundaries between different software components. As the software stack shown in the illustration assumes a monolithic operating system design, the kernel includes device drivers, file systems, memory management, and other subsystems, such as networking. Other kernel architectures in which some or all of these are moved out of the kernel exist, but are less widely used. All software at this level of the stack, running in kernel mode, can be considered to be systems software, as its purpose is to provide the basic resource management and services required by software components higher up in the stack.

Moving up in the stack, the next level comprises system libraries. Such libraries offer basic functionality required by many applications, such as the memory and process management functions provided by libc or the basic mathematical functions found in libm. More complex libraries that provide, for instance, Transport Layer Security (TLS) or compression implementations, also belong at this level of the stack. Software at this layer is also considered systems software, as it provides common functionality to components at the layers above.

Further up in the stack are system utilities and services. This includes, for instance, compilers, linkers, interpreters, or the shell, as well as other software components that are on the line between systems and applications software. While software at this layer is often used directly by users, for instance, when developing

software or performing system administration tasks, it is also frequently depended upon by other application software. Therefore, it is important for such system utilities to be dependable, and we include them in our definition of systems software.

Dependability and security issues at different levels of the software stack can differ in the impact they have on the reliability and security of the overall software system. Components at the operating system level of the stack provide fundamental functionality that virtually every component further up on the stack depends on. Flaws at this level are challenging if not impossible to compensate for higher up in the stack. Dependability shortcomings in the operating system kernel itself can cause system crashes, hangs, performance issues, and numerous other, more subtle flaws. For instance, faulty file systems can lead to data corruption, which can be difficult to detect in data that is not accessed frequently. Security issues at this level are also particularly severe, as they can allow attackers to gain access at the highest privilege level and bypass the security and isolation mechanisms provided by the operating system, which components higher up in the stack depend upon.

Moving up in the stack, flaws in system libraries as well as system utilities and services are generally not as severe as in the OS kernel, but still have the potential to impact each application depending on them. For widely used libraries, this means that flaws or vulnerabilities can impact a large number of different applications. Notable examples of vulnerabilities in systems libraries include the Heartbleed bug in OpenSSL [Syn] and the Stagefright vulnerabilities in libstagefright [MITb] and libutils [MITc] on Android, both of which affected numerous applications relying on these libraries.

Since dependability and security issues in systems software, including both kernel and user mode systems software, can affect all software components higher up in the stack that depend on them, it is particularly important for systems software to be dependable and secure. To ensure that this is the case, dependability and security assessment techniques for systems software are essential.

1.2 Dependability and Security

In this section, we discuss the notions of dependability and security underlying this thesis and introduce relevant concepts and background information.

For dependability, we base our discussion on the taxonomy given by Avizienis et al. [Avi+04], where dependability is defined as "the ability to deliver service that can justifiably be trusted", or alternately as "the ability to avoid service failures that are more frequent and more severe than is acceptable" [Avi+04, p. 13]. The first definition necessitates a way to *justify* trust in a system's ability to deliver correct service, while the second definition requires the system's failure frequency and severity to remain below a specified threshold. In either case, dependability assessment techniques are therefore key to determine whether a system adheres to either definition.

Dependability comprises five different attributes:

1. *Availability*: A system is said to be available if it is ready to provide correct service.
2. *Reliability*: A system is reliable if it continuously provides correct service. While availability refers to a system's ability to provide correct service at any point in time, reliability refers to its ability to sustain that delivery of correct service over a period of time.
3. *Safety*: A system is safe if it does not inflict harm on its users or its environment during normal operation or during system failures.
4. *Integrity*: System integrity refers to the absence of improper alterations, including deliberate as well as accidental alterations.
5. *Maintainability*: A system is maintainable if it is suitable for (proper) modification and repair when required.

Information security is frequently understood to encompass three attributes:

1. *Confidentiality*: A system fulfills the confidentiality attribute if it does not disclose information without authorization.
2. *Integrity*: System integrity refers to the absence of unauthorized alterations.
3. *Availability*: A system is available if it is ready to provide correct service to authorized users.

Together, these three attributes are known as the Confidentiality, Integrity, Availability (CIA) triad. It has long been understood that a violation of any of these three attributes constitutes a potential security violation, which can then be categorized as an unauthorized release of information (a violation of confidentiality), an unauthorized modification of information (a violation of integrity), or unauthorized denial of use (a violation of availability) [And72; SS75].

The attributes of the CIA triad overlap with the attributes of dependability listed above, but the two common attributes, integrity and availability, differ subtly in their definition. In the context of security, availability is specifically only required for *authorized* actions, and integrity refers to the lack of *unauthorized*, as opposed to merely *improper*, alterations to a system.

The taxonomy introduced above further allows for secondary attributes beyond the ones listed above. An example of such a secondary attribute is robustness, defined as "dependability with respect to external faults" [Avi+04, p. 23]. Robustness is a further example of an attribute that falls under both dependability and security.

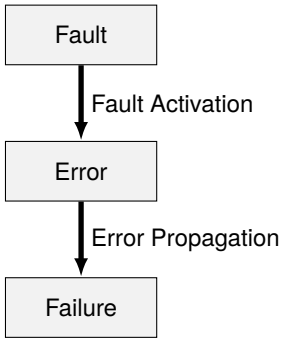


Figure 1.4: The Threats to Dependability and Their Relationship

For instance, a robust system is not susceptible to attempts by an attacker to trigger faults in the system by deliberately providing malformed inputs, or by exposing the system to environmental conditions outside of its specification.

Threats to Dependability and Security

A system operates dependably if it delivers service according to its specification and fulfills the attributes of dependability listed above. The same applies to security, likewise according to the attributes and definitions given above. When a system fails to provide correct service, a service *failure* has occurred. Failures do not arise spontaneously but rather due to a prior deviation within the state of the system. Such a deviation is termed an *error*. However, not every error causes a failure. An error may affect a part of the system state that is not directly related to the provided service, for instance, in parts related to debugging or maintenance, or the affected state may be overwritten before causing a failure. The mechanism by which errors cause failures is termed *error propagation*.

Just like failures, errors do not arise spontaneously and also have an underlying cause. Errors arise when a *fault* in the system is activated (*fault activation*). Faults are flaws, either within or outside the system, in hardware or in software. Software bugs are examples of faults, as are malformed inputs or production defects in hardware. To cause an error, a fault must be activated. For example, in the case of a software bug, fault activation would occur when the faulty code is executed and affects the system state, thereby causing an error.

To summarize, faults, when activated, cause errors, and when an error propagates to a part of the system state affecting the service it provides, a service failure arises. This relationship between faults, errors and failures is illustrated in Figure 1.4.

Faults and failures can all be categorized further. Failures can be categorized by domain, detectability, consistency, and severity of consequences. For faults, Avizienis et al. propose a classification into eight different fault classes, including the phase of creation or occurrence, where they originate relative to the system bound-

aries, and their dimension, that is, whether they originate in software or hardware. They note that most faults belong to three major, overlapping groups:

1. Development faults: All faults that occur or originate during development are development faults. This class includes software bugs or hardware production defects.

2. Interaction faults: All external faults, that is, all faults that originate outside the system boundaries, are interaction faults. This includes, for instance, malformed inputs.

3. Physical faults: All faults that affect the hardware are physical faults.

Of these three groups, the first two are within the scope of this thesis. Physical faults are not covered. The SFI work described in this thesis deals with both development and interaction faults: Considering an entire software system, injecting software faults in individual components simulates the effects of development faults. Injecting faults into a component *A* which interacts with another component *B* can expose *B* to an interaction fault, which is useful for robustness testing of *B*. In this way, SFI can be used to study the effects of both development and interaction faults, depending on which injection locations and system boundaries are chosen. Our work on fuzzing, on the other hand, targets only interaction faults. The malformed inputs produced during fuzzing constitute external faults, and internal faults in the parts of the system parsing or processing these inputs can then lead to errors and, eventually, system failures. As many systems need to robustly handle untrusted inputs, the kinds of faults uncovered with fuzzing techniques frequently have not just robustness but also security implications.

1.3 Research Questions and Contributions

This thesis strives to address the research questions stated below, resulting in the contributions outlined below. The shared goal of all developed approaches and techniques is to improve dependability and security assessment techniques, especially with respect to efficiency. These techniques have been developed with applicability to the lower levels of the software stack in mind and have been tested on software that can generally be categorized as belonging to these lower levels. A substantial portion of the work described in this thesis deals with OS kernel code, as the dependability of kernel code is crucial to the dependability of the overall system. The first three research questions deal with the applicability and efficiency of SFI and Error Propagation Analysis (EPA) techniques, whereas the fourth research question is related to fuzzing.

Research Question 1 (RQ 1): How can the effects of faulty OS kernel components on other parts of the system be identified in the absence of externally visible failures?

Modern OS kernels, even in monolithic designs, consist of numerous interacting components. While some of these components implement core functionality, others, such as file systems or device drivers, are commonly developed and maintained independently of one another and the core kernel components, which can lead to variations in code quality and dependability. Since monolithic systems do not enforce boundaries between such components at runtime, faults in any component can affect other components in various ways, including corruption of their internal state in a manner that may not directly lead to an observable failure. Such state corruption may nonetheless affect the system execution at a later point in time, which is particularly problematic since OS kernels are usually long-running systems. This makes the application of SFI techniques challenging, as the absence of an observable failure is insufficient to decide whether state corruption has occurred and test durations need to be kept short.

Contribution 1 (C 1): A tracing-based approach for assessing the effects of faults in kernel modules.

To tackle the problem outlined above, previous work on SFI has made use of execution tracing to assess the effects of activated faults [APB14; Lan+14; TP13]. However, applying such techniques to components in a monolithic OS kernel is challenging due to the lack of clear component boundaries and well-defined interfaces — potentially, any memory access can constitute a cross-component interaction. Therefore, execution traces need to be gathered on the granularity of memory accesses, which presents further difficulties as user space solutions like Valgrind [NS07a] or Pin [Luk+05] are not applicable to kernel code, and existing kernel tracing solutions like SystemTap [Sys] or LTTng [DD06] do not support tracing at the required granularity. To address these challenges, we develop a scalable, fully automated approach for tracing error propagation in monolithic OS kernels. We describe our approach, called TrEKer, in Chapter 2, which is based on material from [Cop+17]. Our approach makes use of both static and dynamic analysis to assess whether a faulty component has affected other parts of the kernel. We limit tracing overhead by using compile time instrumentation and restricting the instrumentation points to the injection target. Gathered traces are processed and analyzed to determine which parts of a component's behavior are visible to other parts of the kernel. Deviations in such parts constitute potential error propagation. We demonstrate the applicability of our approach by applying it to three different, widely used Linux kernel modules. This contribution has been documented in the publication "TrEKer: Tracing Error Propagation in Operating System Kernels" at ASE 2017.

Research Question 2 (RQ 2): How can SFI experiments be accelerated and adapted to efficiently utilize modern, parallel hardware?

As software grows increasingly complex, the number of SFI experiments necessary for dependability assessment also grows, which can render comprehensive SFI-based dependability assessment of complex software practically infeasible. This concern applies to all parts of the software stack but to systems software in particular as its dependability is crucial to the dependability of the overall system. One way to tackle this issue is to take advantage of the increasing number of execution units available in modern processors and simply run multiple SFI experiments in parallel. However, not only has prior work shown that doing so can impact result validity and may require adjustments to timeout-based detectors [Win+15], parallelization on its own is also insufficient to tackle the problem as it does not address the substantial amount of redundant work that occurs in typical SFI workflows. For every faulty version that is executed in a typical SFI workflow, prior to the point at which the execution reaches the injected fault, the same code is executed with the same inputs as for all other versions.

Contribution 2 (C 2): A technique for accelerating SFI experiments by avoiding redundant work and facilitating parallelization.

We describe a technique to speed up SFI experiments for user mode code in Chapter 3, which is based on material from [Sch+18a]. This technique, called FastFI, makes use of static and dynamic analysis to accelerate SFI tests for user mode code by avoiding the execution of faulty versions containing faults that the workload does not activate, by not redundantly re-executing common execution prefixes for different faulty versions, and by facilitating parallel execution of multiple faulty versions of the same function. FastFI furthermore reduces the compilation time required to generate faulty versions. We forgo heavyweight isolation mechanisms such as Virtual Machines (VMs) to prevent different faulty versions from interfering with one another due to, for instance, the usage of shared resources. Instead, we rely on process-level isolation as this allows us to make use of the fast, lightweight process management functionality provided by modern OS kernels, such as efficient, copy-on-write-based fork implementations, while ensuring that each faulty version is executed in its own address space. We develop a prototype implementation of the approach that targets user mode software written in C. To demonstrate the applicability and performance of FastFI, we apply our prototype to applications from the PARSEC benchmark suite [Pri09]. This contribution has been documented in the publication "FastFI: Accelerating Software Fault Injections" at PRDC 2018.

Research Question 3 (RQ 3): How can SFI and EPA on OS components be accelerated and efficiently parallelized?

As noted above, dependability assessments using SFI are particularly challenging for monolithic OS kernels. In addition to the growing complexity of kernel components such as modern file systems, the lack of well-defined component interfaces and runtime isolation allows faults in any component to arbitrarily affect other parts of the system, necessitating the use of EPA. We have tackled some of the associated challenges with TrEKer (cf. Chapter 2), but the additional tracing requirements further increase test latencies, adversely affecting the feasibility of thorough, SFI-based dependability assessment. Moreover, our SFI acceleration approach FastFI (cf. Chapter 3) is not directly applicable to kernel code as it relies on process management functionality that is only available in user space. Furthermore, FastFI involves a novel execution model that is not directly compatible with TrEKer-style EPA as faulty versions do not execute the entire workload and can therefore not produce complete execution traces. To tackle the issue of long execution latencies for kernel SFI, an approach conceptually related to FastFI but applicable to kernel code and a compatible TrEKer-style EPA approach are required.

Contribution 3 (C 3): An approach to reduce SFI test latencies for OS kernel components while allowing detection of internal state corruption.

Attempting to apply a FastFI-style mechanism for accelerating SFI experiments to kernel code raises several challenges, including the need to quickly and efficiently clone the SUT as well as the required adaptations and enhancements to the EPA approach to support the new execution model. We describe our approach, which is based on material from [CSS20], in Chapter 4. We make use of static and dynamic analysis alongside modern OS, file system, and Virtual Machine Monitor (VMM) features — such as Copy-on-Write (CoW) file copies and VM snapshots — to apply a technique conceptually similar to FastFI to kernel code running in a VM, thereby reducing SFI test latencies by avoiding redundant work and facilitating parallelization. We implement the fast VM cloning required for this approach using off-the-shelf components and without modifying the host kernel or VMM. Moreover, we augment TrEKer, the EPA approach described in Chapter 2, to work with the execution model used in our approach and integrate it with our implementation. We demonstrate the feasibility of our approach by applying it to seven widely used Linux file systems and studying its performance and impact on result validity and EPA results. This contribution has been documented in the publication "Fast Kernel Error Propagation Analysis in Virtualized Environments" which has been submitted to OSDI 2020.

Research Question 4 (RQ 4): Can selective instrumentation of memory accesses characterize program executions in a manner suitable to guide feedback-driven evolutionary fuzzing?

Fuzzing comprises a wide variety of testing techniques that are commonly used for finding security and dependability issues. One class of fuzzing techniques that has proven particularly successful are approaches that use information about the behavior of the SUT during prior executions to decide which inputs should be subjected to further mutation. This is called feedback-driven fuzzing. Most commonly, the feedback mechanism used is based on the control flow path taken by the SUT during prior executions. For instance, inputs which caused the SUT to cover previously unseen edges or new basic blocks are mutated further. However, control flow is not the only way to characterize a program execution. For instance, as shown by our work on EPA with TREKER (cf. Chapter 2), memory access tracing can be a useful way to characterize a program execution and may provide more fine-grained information about the behavior of the SUT. However, straightforward memory access tracing would incur substantial performance impacts, and many memory accesses performed by typical fuzzing targets may be entirely input-independent and therefore provide no useful guidance to the fuzzer.

Contribution 4 (C 4): A technique to use memory access instrumentation to guide evolutionary fuzzing.

It is possible to augment evolutionary fuzzing by using information about input-dependent memory accesses instead of or in conjunction with control flow information. We describe our technique for doing this, called MEMFUZZ, in Chapter 5, which is based on material from [CSS19]. MEMFUZZ uses static analysis to filter out input-independent memory accesses. Memory accesses for which the accessed address may depend on the input provided to the SUT are instrumented at compile time. We use a conservative, intraprocedural static analysis to determine which memory accesses to instrument. Our static analysis and instrumentation are implemented as an LLVM pass. At runtime, a bloom filter is used to store the memory addresses used by the SUT. By using a fixed size data structure, we ensure that we do not need to allocate or free memory to store addresses used by the SUT. We build MEMFUZZ as a modification of AFL [Zal], a widely used feedback-driven fuzzer that normally relies solely on edge coverage as its feedback mechanism, and apply it to three different, widely used target programs. Our results show that different ways of characterizing program executions can guide the fuzzer to find different, distinct crashes. This contribution has been documented in the publication "MemFuzz: Using Memory Accesses to Guide Fuzzing" at ICST 2019.

1.4 Publications

The following publications have, in parts verbatim, been included in this thesis.

[Cop+17] Nicolas Coppik, Oliver Schwahn, Stefan Winter, and Neeraj Suri. "TrEKer: Tracing Error Propagation in Operating System Kernels". In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. ASE 2017. Urbana-Champaign, IL, USA: IEEE Press, 2017, pp. 377–387. DOI: 10.1109/ASE.2017.8115650

[Sch+18a] Oliver Schwahn, Nicolas Coppik, Stefan Winter, and Neeraj Suri. "FastFI: Accelerating Software Fault Injections". In: *23rd IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*. PRDC 2018. Taipei, Taiwan, Dec. 2018, pp. 193–202. DOI: 10.1109/PRDC.2018.00035

[CSS19] Nicolas Coppik, Oliver Schwahn, and Neeraj Suri. "MemFuzz: Using Memory Accesses to Guide Fuzzing". In: *12th IEEE International Conference on Software Testing, Verification and Validation*. ICST 2019. Xi'an, China, Apr. 2019, pp. 48–58. DOI: 10.1109/ICST.2019.00015

[CSS20] Nicolas Coppik, Oliver Schwahn, and Neeraj Suri. "Fast Kernel Error Propagation Analysis in Virtualized Environments". In: *14th USENIX Symposium on Operating Systems Design and Implementation*. OSDI 2020. 2020. [under submission]

The following publications are related to different aspects covered in this thesis, but have not been included.

[Sch+19] Oliver Schwahn, Nicolas Coppik, Stefan Winter, and Neeraj Suri. "Assessing the State and Improving the Art of Parallel Testing for C". in: *28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2019. Beijing, China: ACM, 2019, pp. 123–133. DOI: 10.1145/3293882.3330573

[Sch+18b] Oliver Schwahn, Stefan Winter, Nicolas Coppik, and Neeraj Suri. "How to Fillet a Penguin: Runtime Data Driven Partitioning of Linux Code". In: *IEEE Transactions on Dependable and Secure Computing* 15.6 (Nov. 2018), pp. 945–958. DOI: 10.1109/TDSC.2017.2745574

1.5 Organization

The rest of this thesis is structured as follows: We address our first research question in Chapter 2, where we develop an approach for determining the effects of faulty components in monolithic OS kernels in the absence of observable failures. In Chapter 3, we address the second research question and describe a technique for accelerating SFI experiments of user mode software. We then turn to the applicability of such a technique for kernel code in Chapter 4, where we address the third research question and describe our approach for accelerating SFI experiments and EPA for kernel code. Chapter 5 addresses the fourth research question and describes our work on memory access instrumentation-guided fuzzing. Finally, concluding remarks, along with a summary of the contributions and key insights, can be found in Chapter 6.

2 Kernel Error Propagation Analysis

Modern operating system kernels consist of numerous separate but interacting components, many of which are developed and maintained independently of one another. In monolithic systems, the boundaries of and interfaces between such components are not strictly enforced at runtime. Therefore, faults in individual components may directly affect other parts of the system in various ways. SFI is a testing technique to assess the resilience of a software system in the presence of faulty components. Unfortunately, SFI tests of operating system kernels are inconclusive if they do not lead to observable failures, as corruptions of the internal software state may not be visible at its interfaces but nonetheless affect the subsequent execution of the OS kernel beyond the duration of the test.

In this chapter we present TrEKer, a fully automated approach for identifying how faulty kernel components affect other parts of the system. TrEKer combines static and dynamic analyses to achieve efficient tracing on the granularity of memory accesses. We demonstrate TrEKer's ability to support SFI oracles by accurately tracing the effects of faults injected into three widely used Linux kernel modules. The contents of this chapter are, in parts verbatim, based on material from [Cop+17].

2.1 Overview

Complex modern software systems generally consist of many interacting components. In larger systems, these components may be developed or maintained by different teams of developers and may differ in numerous aspects, including code quality and the amount of residual faults. In an empirical study of the Linux kernel, Palix et al. find that device drivers, along with file systems and architecture-specific code, have one of the highest rates of faults per line of code in the kernel. As device drivers have a high fault rate and comprise a large amount of code, they are the part of the kernel containing the most faults in absolute numbers, followed by file systems and architecture-specific code. In order to assess the resilience of the overall system, it is necessary to understand how it is affected by individual faulty components, such as, in the case of a monolithic kernel, faulty device drivers or faulty file systems. This is called *error propagation* in the taxonomy by Avizienis et al. [Avi+04], as outlined in Chapter 1. For this purpose, SFI, the deliberate introduction of faults in specific components to simulate their behavior in the presence of residual software faults, is an established approach [APB14; DM06; Nat+13].

In SFI tests, the SUT is exposed to erroneous behavior of a component it is interacting with. That component is termed the *injection target*. Software fault injections are similar to mutations for mutation testing, but commonly based on different assumptions regarding the types and distributions of the introduced faults (see [JH11a; NCM16] for an overview of common fault assumptions in either application). After the injection, interactions between the SUT and the injection target are triggered by a test *workload*. To assess error propagation from the injection target to the SUT, the behavior of the SUT is observed while it is processing the workload to identify behavioral deviations in response to the injection. Unfortunately, oracles of this type are generally insufficient to make any conclusions whenever no such behavioral deviations are observed. In such cases, there are three different possibilities:

1. The fault has not been activated,
2. The fault has been activated, but its effects have not propagated to the user interface, therefore there are no observable behavioral deviations.
3. The fault does not affect the system behavior, for instance because the mutated version is semantically equivalent to the non-mutated version, either in general or under the specific workload used for the test.

While the first case can be identified by additional code that logs the activation of injected faults, which can easily be added when performing SFI, distinguishing the latter two requires observing the internal state of the SUT during test execution. SFI test frameworks commonly use execution trace comparisons across setups with and without injected faults as a secondary oracle to distinguish between these cases [APB14; Lan+14; TP13].

2.1.1 The SFI Oracle Problem for OS Kernels

While the gathering and comparison of execution traces alleviates the aforementioned oracle problem, applying it is challenging for an important class of SUTs: OS kernel components in monolithic OS kernels, where all kernel components are interacting within the same address space and with the same privileges. Without memory protection between kernel components all memory is shared and directly accessible. This makes every memory operation in the system a potential cross-component interaction affecting the SUT, meaning that all such memory operations need to be traced. Existing memory tracing approaches for user space applications (e.g., using Valgrind [NS07a] or Pin [Luk+05]) are not applicable for OS kernels. Existing tracing approaches for OS kernels (e.g., SystemTap [Sys] or LTTng [DD06]), on the other hand, only provide tracing on the granularity of function calls instead of individual memory accesses. A naïve tracing of *all* memory operations is infeasible,

as the kernel code base is large and some parts, such as hardware interrupt handling routines, are performance critical. Simply attempting to instrument and log every memory access in the kernel would therefore incur prohibitive performance overhead, adversely affect timing behavior, and cause issues when the logging code itself relies on parts of the kernel that been instrumented.

2.1.2 TrEKer: Solving the SFI Oracle Problem

To correctly identify and characterize the effects of residual software faults in kernel components, we present TrEKer, a scalable, fully automated approach for Tracing Error propagation in operating system Kernels that relies on a combination of static and dynamic analyses to infer error propagation from a faulty kernel component to other parts of the kernel. TrEKer limits the trace points to the injection target and infers error propagation from deviations in the injection target's state and behavior that are potentially visible to other parts of the kernel, thereby effectively improving the soundness of SFI tests for OS kernels at the cost of execution time overheads for trace collection and analysis.

We demonstrate TrEKer's ability to trace the effects of faults in three widely used kernel components on the Linux kernel. We find that up to ~10 % of seemingly successful runs in our fault injection experiments would be misclassified by conventional oracles.

The remainder of this chapter is organized as follows: Section 2.2 gives an overview over related work. Our proposed approach is detailed in Section 2.3. We discuss TrEKer's implementation in Section 2.4 and the experimental analysis in Section 2.5. Concluding remarks for this chapter can be found in Section 2.6.

2.2 Related Work

To classify the results of SFI tests on kernel code, TrEKer traces the effects of injected software faults in OS kernels. We discuss existing trace-based approaches for user mode software in Section 2.2.1, alternative approaches for kernel-level SFI tests in Section 2.2.2 and trace comparison in Section 2.2.3.

2.2.1 Execution Trace Based Oracles for User Mode Software

Execution tracing has been widely adopted to determine the outcome of SFI tests [Aid+01; APB14; CMS98; Lan+14; NCM16; Pip+12; Pip+15; TP13], for reasons similar to our motivation for TrEKer. In such approaches, execution traces of the unmodified SUT are recorded and later used as a *golden run* oracle to compare executions with injected faults against. The techniques used to record execution traces can be broadly classified as belonging to three different categories.

One class of approaches (e.g., [Aid+01; CMS98; Pip+15]) uses debuggers to record execution traces. This imposes execution latencies that are not tolerable by many SUTs, among them the OS kernels targeted by our work. Interrupt service routines, for instance, need to have short response times and exceeding those due to the overhead imposed by using a debugger to gather traces may result in unintended OS failures during the SFI test.

A second class of approaches (e.g., [Sch+15; SPS09]) uses full-system simulation for execution tracing. Full-system simulators implement the semantics of low-level hardware operations for a given target platform in software. The SUT is executed on this simulated hardware model. Although the simulation of every single hardware operation in software imposes massive execution time overheads, this is not observable by the SUT itself. Any latencies observable by the SUT are based on the simulated hardware model, including cycle counts or simulated hardware timers. Therefore, full-system simulators are generally suitable for tracing OS kernel executions, but massively impair test throughput due to the simulation overhead.

The third class of approaches (e.g., [Lan+14; ZKB13]) relies on Dynamic Binary Instrumentation (DBI) or Dynamic Binary Translation (DBT). Such approaches typically use tools or frameworks such as Pin [Luk+05], Valgrind [NS07b] or DynamoRIO [BGA03]. Similar approaches have been developed for OS kernels [BL07; FBG12; Hen+14; KB13], but none of them have been used for execution tracing in SFI tests. As TrEKer instruments kernel code during compilation, it is independent from kernel modules that need to co-evolve with changing kernel interfaces. In this respect it differs from the work of Feiner et al. [FBG12] or Kedia and Bansal [KB13]. Moreover, approaches based on binary translation only work for the specific hardware architecture or architectures supported by the chosen framework and require adjustment for others. PinOS [BL07], for instance, is limited to IA-32. The applicability of TrEKer, in contrast, is not limited to any specific OS kernel or hardware architecture, as long as the instrumentation target can be compiled for that architecture with Clang/LLVM. As both PinOS and DECAF [Hen+14; Hen+16] rely on virtualization, they cannot be applied for hardware-specific kernel code, such as device drivers, if that hardware cannot be emulated by the underlying hypervisors.

2.2.2 Oracles for Kernel-Level SFI Tests

Due to the SUT and architecture specificity of available kernel tracing tools, SFI tests for these SUTs commonly employ other, less accurate oracles.

Koopman et al. have introduced a classification of OS failure modes that they consider relevant and implemented corresponding detectors in the Ballista project [Koo+97]. Their classification comprises five different failure modes, collectively referred to as the "CRASH scale", where each letter of the acronym stands for a failure mode:

- **C**atastrophic failures are failures that render the entire system unusable, e.g., kernel panics or blue screens.
- **R**estart failures are cases where the OS silently stops responding to requests made by the executing test case, also known as a system hang, which requires a restart to resolve.
- **A**bort failures are cases in which the OS detects a problem and responds by notifying the executing test, e.g., by signaling a segmentation fault.
- **S**ilent failures denote the violation of the kernel's specified behavior without corresponding notification to the executing test.
- **H**indering failures are failures that mislead debugging efforts, e.g., by returning an incorrect error code.

Arguing that these are the most critical failure classes, Ballista and similar approaches to OS robustness testing limit their oracles to the detection of the first three classes of the CRASH scale [FX02; IVD15; JZ08; KKS98; Reg05].

In contrast, TrEKer focuses on the detection of silent or "non-crashing" failures, such as Silent Data Corruption (SDC), which constitute a significant threat to reliability [Cot+15; Lo+09; Lu+14; ZE13] and have been largely ignored by prior work on OS level SFI tests. The reliable detection of *restart* failures requires kernel execution traces containing every single executed instruction. While TrEKer is capable of implementing such a tracing policy, the required heavy-weight instrumentation may result in performance degradation similar to the approaches discussed in Section 2.2.1. We, therefore, limit the scope of our approach to the detection of error propagation in the case of terminating test executions and employ existing timeout-based detectors for *restart* failures.

While a number of tools (e.g. SystemTap [Sys] and LTTng [DD06]) exist to trace the execution of OS kernel code using probes (cf. [Cot+13; KIT93a] for SFI tracing), they are only capable of tracing function invocations and not individual memory accesses. To identify how faults affect SUT state, i.e., the data the SUT operates on, TrEKer selectively instruments memory operations that are invisible to these tools.

2.2.3 Trace Comparison

To detect error propagation, TrEKer compares traces of executions with injected faults to golden run traces of the unmodified SUT. Trace comparison is also commonly used for fault localization. Wong et al. give an extensive overview [Won+16]. Such approaches typically compare traces of the same version of the SUT with different inputs to identify the root causes of behavioral divergences. Therefore, they are not directly applicable to the scenario targeted by TrEKer.

2.3 System Model

We propose an approach for identifying how faults in components in a monolithic OS affect the rest of the system. To that end, this section starts with a brief overview of the underlying fault taxonomy in Section 2.3.1, followed by a discussion of the systems we consider and their component interactions in Section 2.3.2.

2.3.1 Faults and their Consequences

As noted in Section 2.1, we follow the fault taxonomy by Avizienis et al. [Avi+04]. We discussed this taxonomy in Section 1.2, so we limit ourselves to a brief recap of the relevant concepts here. Any system or system component[1] is assumed to implement a *system function* according to a *functional specification*. The system implements the system function as a sequence of *states*. The fraction of a system state that is perceivable at the system's *interface* is called the *external state*. The sequence of external states implementing the system function is referred to as *service* and the deviation of service from the functional specification is called a *failure*. The deviation of an external state in the sequence that constitutes the service may be caused by a prior deviation of the system's internal state that is indistinguishable at the interface from a correct implementation of the system function. Such a deviation of the system state is called an *error*. The cause of an error is termed as a *fault*. By these definitions, a fault is "something that possibly leads to an error", an error "something that possibly leads to a failure", and a failure "a deviation of observed behavior from specified behavior."

When a fault causes an error, this is referred to as *fault activation*, and the effect an error has on subsequent system states is called *error propagation*.

2.3.2 Monolithic Operating Systems and Composition

We assume that component-based, monolithic OS kernels can be seen as follows:

- There is a core part which provides essential functionality, such as basic process and memory management and is, therefore, always necessarily present.
- All functionality outside of the core part is implemented by an arbitrary number of modules, including device drivers and file systems.

Modules can interact with one another or the core kernel through function calls, thereby explicitly exchanging information via parameters and return values. Furthermore, the system does not enforce any memory isolation between its components. All modules and the core kernel share the same memory address space and

[1] We mean "system or system component" whenever we refer to "system" in this subsection.

can, in theory, freely access and modify each others data structures. Finally, modules can also access and modify any global data structures in the system. This lack of isolation allows for implicit component interactions and exchange of information outside of the explicit mechanism of function calls and return values.

Although the implementation we describe in Section 2.4 utilizes runtime loading and unloading of kernel modules, the fundamental approach described here does not conceptually rely on the availability of this functionality. TrEKer is equally applicable to a kernel that does not provide this functionality.

Due to the lack of runtime isolation or protection mechanisms, a faulty module can affect other modules or the core kernel in a variety of ways. In particular, tracing mechanisms that only consider parameters and return values of function calls cannot capture differences in communication through shared memory. However, due to the aforementioned lack of isolation, distinguishing between memory accesses that constitute potential shared memory communication, particularly write accesses by a faulty component, and those that do not is usually not straightforward without examining the entirety of all other modules and the core kernel. However, the result of such an analysis would be dependent on the particular modules present in the system in question and changes to this configuration might well yield different results. In practical terms, it would also incur substantial overhead and potentially cause problems with performance or timing sensitive parts of the kernel. Therefore, we limit our analysis to the faulty component itself and analyze which fraction of its state can be expected to be accessible by *any* other component in the system, independent from the actual system configuration. We denote this fraction of expected externally visible behavior as the component's *interface*. The interface includes parameters of function calls from and return values of function calls to the component as well as memory accesses to locations that can reasonably be assumed to be used for transmitting data to other components via shared memory. We detail in the following what we do and do not consider such interface relevant memory accesses within a component targeted by our analysis.

Read accesses are not generally considered part of the component interface. If the value that is read was previously written by the faulty module itself, the read access is a purely internal operation and clearly does not constitute external communication. If, on the other hand, the value was written by another module, we do not consider the read access itself to be behavior visible to other components: While a faulty module may attempt to read from the wrong address, resulting in an unexpected value, this does not directly result in externally visible differences in behavior. Cases where it has indirect influence (for instance, if the faulty module proceeds to use the wrong value as a function parameter) will be captured under our notion of interface at the point where the behavior in question becomes externally visible. Finally, cases where a faulty module attempts to access an *invalid* address, such as a null pointer, do potentially constitute externally visible behavior, but are already

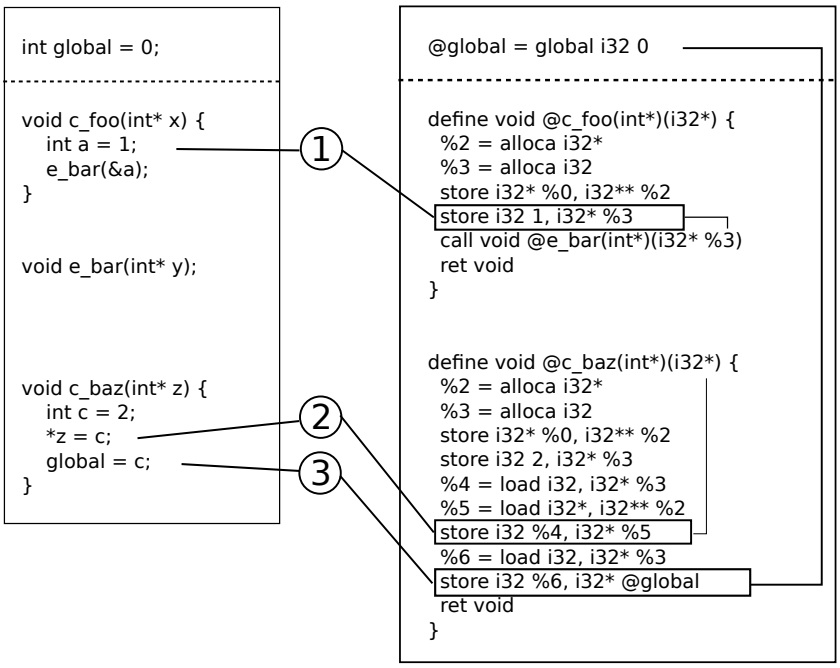


Figure 2.1: Three different cases in which write accesses can be externally visible, in C and simplified LLVM IR.

captured by existing error detectors and typically result in a system failure, such as a kernel panic.

Local write accesses are store operations to addresses that are not known to any components other than the faulty one. Most notably, this includes accesses to stack-allocated local variables unless their address is passed to another component (either directly, e.g., as a function parameter or implicitly by writing it to another externally visible memory location). Access to regions of memory that are allocated and freed without ever being referenced in an externally visible manner (that is, as with stack addresses, passed to external functions or written to externally visible addresses) in between also fall into this category. Such accesses are not considered externally visible for the purposes of our analysis and are therefore not deemed part of the component interface.

Externally visible write accesses are store operations to addresses that are known to components other than the faulty one. This includes all addresses that are passed to the module from another component, for instance as a parameter or return value, as well as globals and addresses belonging to memory that has been allocated by the module itself but then communicated to other components. As mentioned above, it also includes all memory addresses that are reachable by following pointers from another externally visible address.

Of these three categories, the one most relevant for our analysis is the last one, externally visible write accesses. We distinguish between three different cases of external visibility of write accesses:

1. Writes to an address that the component passes to another component or to addresses that are reachable from such an address;
2. Writes to an address that was passed to the component by another component or to addresses that are reachable from such an address;
3. Writes to global variables or to addresses that are reachable from a global variable.

These three cases are illustrated in Figure 2.1. In the first case, a function in the component (`c_foo`) writes to a variable (`a`) and then passes the address of that variable to an external function (`e_bar`). Should the external function dereference that address, the result of the written value would be accessible to it. Note that we do not inspect whether such an access actually occurs as that would require instrumenting the external component itself with the associated drawbacks discussed above, we just check whether it is possible. In the second case, a function in the component (`c_baz`) has received a pointer (`z`) as an argument and writes to that address. If the caller of `c_baz` is an external function (e.g., `e_bar`), that write access is visible to that caller. In the final case, a function in the component writes to a global variable (`global`). As `global` is visible to all components in the system, this write access is also externally visible.

In all of these examples, the external visibility of the stores in question is fairly straightforward to recognize, requiring at most one pointer dereference. However, more complicated cases exist, for which we introduce the following notion of reachability: An address `p` is *directly* reachable from an address `q` in the following cases:

1. `p` is stored at `q` (i.e. `*q = p`).
2. `q` is the base address of a data structure (e.g., an array or struct) and `p` is the address of a member of that data structure (e.g., `p = &(q->foo)`).

An address `p` is *indirectly* reachable from an address `q` if there is an address `r` such that `p` is directly reachable from `r` and `r` is reachable (either directly or indirectly) from `q`.

We consider the externally visible behavior of a component at its interface with the rest of the operating system to consist of the values of parameters passed to functions outside the component, the values returned to callers outside the component, the externally visible memory addresses it writes to and the values it writes to them.

Error propagation occurs when a faulty component exhibits externally visible behavior that a fault-free version of the same component will never exhibit under the same workload.

2.4 TrEKer: Tracing Error Propagation in OS Kernels

The implementation work required to realize our proposed approach comprises two essential parts: An instrumentation tool capable of gathering the information required to fully capture the externally visible behavior of a target component and an analysis tool to perform the filtering and transformations required to distinguish between the cases described in Section 2.3. We describe these parts in Section 2.4.1 and Section 2.4.2, respectively. Trace comparison is described in Section 2.4.3.

2.4.1 Component Interface Identification and Instrumentation

The purpose of the instrumentation phase is to gather all the information required to reconstruct an accurate model of the externally visible behavior of the target component. To that end, the instrumentation needs to capture the addresses and values of memory accesses as well as function call targets, parameters and return values. Function call instrumentation needs to be performed both on the caller side, when the target component calls functions in other components, as well as the callee side, when other components invoke functions of the target component. This is necessary to determine when control flow enters or exits the target component, and to track the associated parameters and return values.

In order to avoid limiting TrEKer to a specific OS or architecture, we have decided to implement compile time instrumentation as an LLVM [LA04] optimization pass, allowing us to support native execution on a various different architectures.

As an LLVM optimization pass, the instrumentation step operates on LLVM IR, a Static Single Assignment (SSA) representation. Unlike x86 assembly, only a small number of LLVM instructions operate on memory, most notably the `load` and `store` instructions. In addition to the memory accesses themselves, the instrumentation also needs to capture accesses to fields of data structures, or more specifically the computation of their addresses based on the base address of the data structure. In LLVM, this is typically modeled by the `getelementptr` instruction.

Furthermore, the instrumentation should capture basic tracing information, such as function entry and exit, arguments and return values. Therefore, it also handles function calls (caller-side), function entry and function exit (callee-side).

Finally, TrEKer is designed for OS kernel components, necessitating a way to instrument inline assembly which is common in kernel code. Attempting to parse and process inline assembly directly suffers from many of the same drawbacks that make binary instrumentation an unattractive choice for kernel tracing, including lack of portability across different architectures and significant added complexity. We would lose the portability advantages gained by choosing to build our implementation on LLVM. In practice, and particularly in the Linux kernel, we find that inline assembly is usually specified using extended inline assembly syntax. Such extended inline assembly statements take a list of input and output variables and

clobbers. The instrumentation can rely on these arguments and constraints to extract which memory addresses may be read from or written to by the inline assembly without parsing it directly. Based on this information, instrumentation can be performed as it would be for `load` or `store` instructions. This allows us to retain the portability enabled by LLVM without foregoing instrumentation of assembly code entirely.

For each of the instrumentation points identified above, the instrumentation pass inserts a function call with the first argument indicating its type. The subsequent arguments differ for the different types of instrumentation points. In addition to memory addresses and values, the information passed to the function also includes static type information (e.g., whether a value is of a pointer type) and hashes of global variable names where applicable. This way, later analysis steps (e.g., the trace analysis described in Section 2.4.2) can identify pointer values in the trace without having to rely on heuristics, such as checking whether a value belongs to a previously seen address range (as in [Lan+14]).

The instrumentation pass inserts calls to a function `inst_wrapper`. For our experiments on Linux kernel modules, we applied a patch to the Linux kernel that implements a stub for this function and a kernel module that, once loaded, handles the logging at instrumentation points. Prior to loading this runtime kernel module, the kernel stub is effectively a no-op, allowing the instrumented module to function even when the runtime has not been loaded. This lets us measure the overhead of instrumentation on its own, separately from the cost incurred by the actual logging. For other application scenarios, such as user-level code, different implementations of the runtime, for instance in a library, would be possible as well.

When the runtime module is loaded, it sets up a function pointer which is then used by the `inst_wrapper` function to call the actual logging implementation. That implementation uses `printk` to output information at each instrumentation point in order to enable reliable tracing even in cases where the target may crash. For other use cases, a trivial performance optimization would involve caching trace data in memory to reduce the number of calls to `printk`, or relying on other mechanisms to transfer logging data from the kernel.

2.4.2 Trace Analysis

We have implemented a trace analysis tool that is capable of performing the reachability analysis for externally visible write accesses that we have described in Section 2.3 as well as deriving symbolic values for the addresses of memory accesses in order to facilitate comparisons between traces. We first describe our implementation of the reachability analysis, followed by our symbolic address generation.

Reachability

In Section 2.3, we have introduced a notion of reachability which incorporates both reachability through pointers as well as through access to member fields of data structures. Our implementation of reachability analysis applies this notion to individual execution traces. First, we split the trace based on the function calls it contains. For each function call, we extract the caller, arguments, return value and called functions. For calls to internal functions, we additionally extract the trace entries generated during that function call. This results in a tree structure in which nodes represent dynamic instances of function calls and edges represent a caller-callee relationship.

Next, we perform the aforementioned reachability analysis for each of the three different cases in which stores performed by the instrumented component may be externally visible.

For the first case, writes in the component that are reachable from arguments passed to an external function, we first identify each node representing a call to an external function taking at least one pointer argument in the aforementioned tree. Then, for each such node, we iterate backwards over the preceding trace entries until we encounter another node representing an external function call. During this traversal, we build up a separate graph which we term the *reachability graph* from the encountered trace entries as follows:

- For load or store entries, check if the address node has an outgoing edge representing a previously seen load or store from that node, and if so, skip this trace entry. This way, we only take the most recent load or store into account. Otherwise, if the read or written value is a pointer, add an edge from the address node to the value node.
- For data structure member access entries (i.e. `getelementptr` instructions in LLVM IR — we omit other forms of pointer arithmetic at this stage), add an edge from the source (i.e. base address) to the destination (member address) node.

Non-existent address nodes are created on demand during the construction of the reachability graph.

In this reachability graph, we identify the set of nodes (addresses) that are reachable from any of the nodes representing pointer arguments passed to the external function. This set of addresses is a subset of the addresses that are visible to the external function, and for each of these addresses, the last write access is deemed visible to the external function. An illustration of a graph for this case can be seen in Figure 2.2: The stack-allocated struct `s_baz` is accessible via the pointer `p` passed to the external function and both of the stores to its members are visible to the called function.

For the second case, writes in the component to global addresses or addresses reachable from them, we iterate over the trace, constructing a reachability graph as follows:

- For load or store entries, if the address node already has an outgoing edge representing a read or write access, remove it so that we only take the most recent load or store into account. Add an edge from the address node to the value node.
- For data structure member entries, add an edge from the source to the destination address.
- Mark global addresses when they are encountered and annotate the corresponding node.

In this reachability graph, we identify the set of nodes that are reachable from any node annotated as representing a global variable. As in the first case, write accesses operating on any of these addresses are deemed visible to the external function.

For the third case, writes in the component that are reachable from arguments passed by an external function, we identify each node representing a call to an external function that in turn calls functions provided by the component. This corresponds to any external function node (including the root node) in the tree that has internal function child nodes. Then, for each component function (that takes at least one pointer argument) called by such an external function, we iterate over all trace entries belonging to that function node and its child nodes, performing an in-order traversal of a sub-tree with the component function at its root. The traversal is stopped when we encounter another external function node. During this traversal, we once again build up a reachability graph, in the same manner as for global addresses, apart from annotating nodes representing global variables. We identify the set of nodes that are reachable from any node representing a pointer argument passed by the external function, and as in the previous cases, deem the last write accesses to any of these addresses visible to the external function.

Symbolic Addresses

To compare traces from different executions, where absolute addresses may differ, a mechanism to map concrete addresses to symbolic addresses is required. We generate symbolic addresses from reachability graphs similar to the ones described previously. A symbolic address consists of an *anchor point*, and a path starting from that anchor point. For writes in the component that are reachable from arguments passed by an external function and writes that are reachable from a global value (the second and third cases discussed above), symbolic addresses use the argument or the global variable as the anchor point and the shortest path from there to the

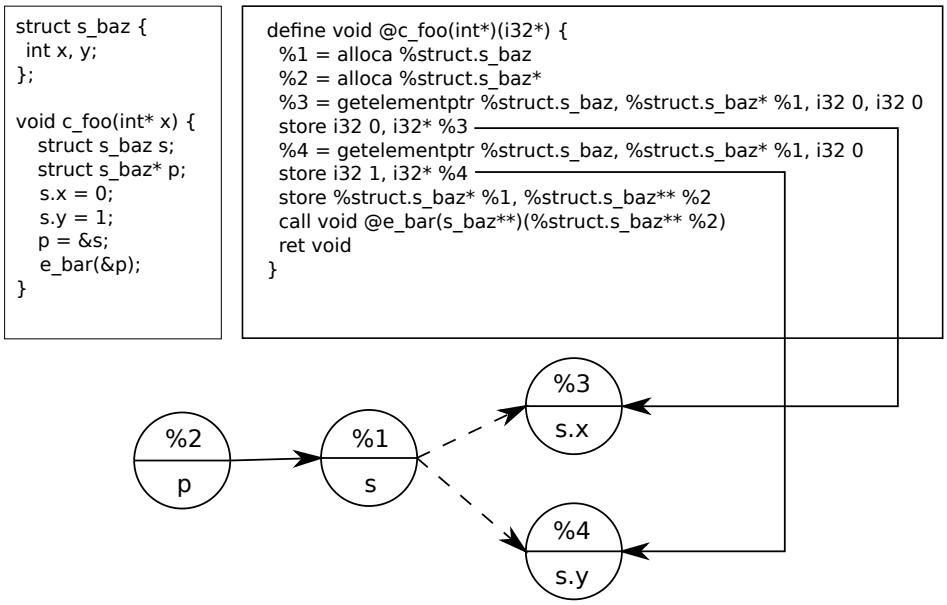


Figure 2.2: An example of a reachability graph and the corresponding code snippet. Solid lines indicate values stored at an address, dashed lines indicate offset calculations. Writes to `s.x` and `s.y` are visible to `e_bar`.

address that was written to as the path. If, for instance, a pointer x is passed to the component, and the component writes to an address y that can be obtained by dereferencing x and adding an offset k, the resulting symbolic address is $x \xrightarrow{*} \xrightarrow{k}$. For writes in the component that are reachable from arguments passed to an external function, symbolic addresses are created using a similar mechanism. In this case, however, a set of anchor points consisting of return values of external functions, stack allocations, global variables and the results of pointer arithmetic is considered. If an address is reachable from several anchor points, we compare the lengths of the shortest paths from each anchor point to the address and pick the shortest one. The same symbolification is performed for values of pointer types.

2.4.3 Trace Comparison

Assessing the impact of faults on visible write accesses requires a mechanism for comparing traces of executions with activated faults to fault-free executions (golden runs). Moreover, in order to minimize the impact of non-deterministic runtime behavior, we need to compare a faulty execution to a set of several fault free runs. While this allows us to more precisely extract those differences between traces that result from the activation of a fault (i.e. behavior that a fault-free implementation would never exhibit), it also complicates the comparison process. We compare traces using the following two-step approach:

TRACE MERGING

First, a set of traces from fault-free runs is processed in order to generate a merged trace structure containing information about the addresses *any* execution writes to as well as the addresses *all* executions write to, along with the corresponding values:

Let t_1 and t_2 be traces from two fault-free executions, both of which consist of the same sequence of function calls and write to address a_1, with the values being v_1 in t_1 and v_2 in t_2. The resulting merged structure then contains a write access $a_1 \leftarrow \{v_1, v_2\}$. Furthermore, let t_1 also write to address a_2. The merged structure then contains, separately, the set of addresses that all executions have written to ($A_{\text{all}} = \{a_1\}$) as well as the set of addresses that at least one trace has written to ($A_{\text{any}} = \{a_1, a_2\}$).

Let t_3 be a trace from a third fault-free execution which consists of a different sequence of function calls. The addresses and values written by t_3 are stored separately from those of t_1 and t_2. In order to support workloads which exercise the target module using multiple threads or processes, merged structures are stored separately for different threads and processes.

TRACE COMPARISON

Next, this merged structure is used in a comparison with a faulty execution. Let t_f be a trace from such an execution. First, the threads or processes in t_f need to be matched to their counterparts in the merged structure. Since absolute thread or process IDs may differ between executions, they do not form a reliable foundation for such a mapping. They are also not applicable for kernel code that does not run in process context, such as interrupt service routines. Instead, we perform the mapping by call sequence, looking first for exact matches. In cases where no exact match is found, the trace exhibiting the previously unknown call sequence can either be ignored so as to avoid introducing false positives, or a best effort comparison with the known call sequence with the longest common prefix can be performed. We call the former option *strict mode* as it provides stronger safeguards against false positives. In the latter option, situations may arise in which several known call sequences have the same longest common prefix length with the new call sequence. In this case, we compare with all of them and report the results for the case in which we discover the fewest divergences. Best effort comparisons are only performed over the common prefix so that we never compare store visibility for different functions. The trace comparison then checks for three different cases:

1. Is each address a_f that t_f writes to also written to in at least one fault-free execution? In case it is not, the write to a_f is deemed an *additional* write access.

2. Does t_f contain writes to all addresses a_i that *every* fault-free execution writes to? If it does not, a write to such an a_i is deemed *missing*.

3. For the set of addresses that both the faulty and at least one fault-free execution write to, is value v_f written by t_f contained in the set of values written by the fault-free executions? If v_f is not in that set, the write access *differs* from the corresponding write accesses seen in fault-free runs.

Non-pointer values are not assigned symbolic counterparts during trace processing but may in some cases take on different values even during most fault-free runs. This can be the case with, for instance, addresses that are written as non-pointer types, timestamps or random values. In order to minimize the number of false positives introduced by such cases, the comparison between faulty and fault-free runs ignores any values that differed in a majority of fault-free runs (i.e. for which the number of observed values is greater than half the number of fault-free runs) when performing the third check above.

The numbers of missing, additional and differing stores are gathered separately for the three cases of write access visibility, resulting in a total of nine combinations.

2.5 Experimental Analysis

In this section, we evaluate our approach by performing experiments with real-world Linux kernel modules: a storage device driver and two file systems. The research questions that we strive to answer in this evaluation are detailed in the following Section 2.5.1. We then describe our SUT in Section 2.5.2. Section 2.5.3 covers the injection targets and Section 2.5.4 our choice of workload. We report on our experimental results in Section 2.5.5.

2.5.1 Research Questions

In order to assess the suitability of TrEKer for SFI experiments on real-world kernel code, we investigate the following research questions:

RQ 1 Is TrEKer a sound detector for error propagation?

RQ 2 Does TrEKer improve the soundness of SFI tests?

RQ 3 What are the overheads resulting from TrEKer's instrumentation?

RQ 4 Does TrEKer's instrumentation affect SFI test results?

2.5.2 SUT

Although TrEKer supports native execution, we perform our evaluation in a virtualized environment to avoid frequent hard machine restarts due to system crashes resulting from the tests. The toolchain we use in the experiments is illustrated in

Figure 2.3. The guest system is Debian 8.6 running in QEMU 2.6.0. It is configured with one CPU and 1 GiB of RAM and has virtual SCSI and NVMe devices attached. KVM is enabled. The guest kernel is Linux 4.4.25, patched to support compilation with Clang/LLVM (using a modified version of the patch set created by the defunct LLVMLinux project) and compiled with Clang/LLVM 3.9.1. The host system is Debian 8.5 running the distribution-provided 3.16 kernel. All experiments are performed using four parallel QEMU instances running on a host system equipped with an i7-4790 CPU and 16 GiB of RAM. Experiment control and timeout detection are handled by a controller running on the same host. The timeout value for all tests is 45 seconds, excluding boot and setup time. In addition to the timeout mechanism, we employ detectors operating on the serial output of the guest system to detect error messages from the kernel. Our detectors distinguish between five different classes of kernel error messages (Call Trace, GPF, BUG, Oops and Panic). We also check exit codes during the workload execution to detect workload failures that did not result in kernel error messages, resulting in a total of eight different experiment result classes.

2.5.3 Injection Targets and Faultload Selection

We apply our proposed approach to three different, widely used Linux kernel modules:

1. `f2fs`, the Flash-Friendly File System, a file system specifically designed for NAND Flash-based storage devices;
2. `btrfs`, a copy-on-write file system implementing various advanced features; and
3. `nvme`, the kernel module providing support for NVMe devices.

For each of these modules, we perform the following series of steps:

1. We inject software faults using the SAFE tool [Nat+13] with default settings,
2. build the resulting module using our compile time instrumentation tool (Section 2.4.1),
3. execute a workload that utilizes functionality provided by the module, and
4. observe the resulting effects during execution and via memory trace comparison.

SAFE performs fault injection at the source code level using the G-SWFIT[DM06] fault operators. We build and instrument the target modules with Clang/LLVM 3.9.1.

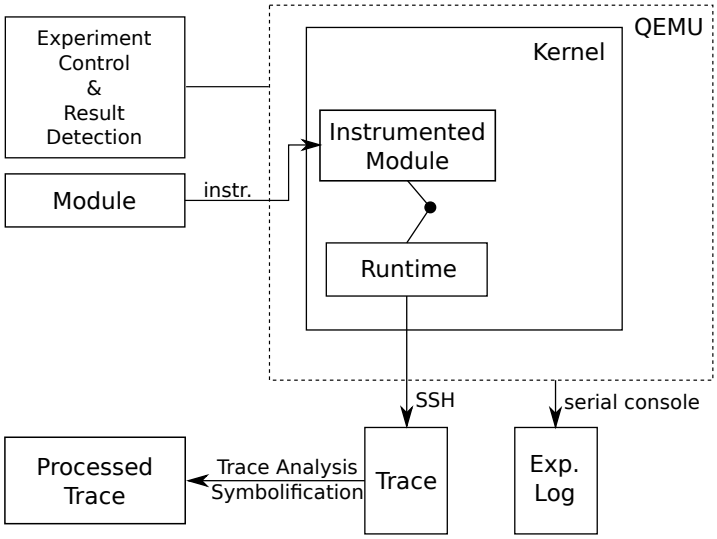


Figure 2.3: The QEMU-based virtualized test environment and toolchain

We use prefix matching for `btrfs` and `f2fs` to maximize the usage of recorded memory traces (see Section 2.4.3). As `nvme` directly interfaces with the system hardware and, thus, has a higher exposure to non-determinism, we use the strict mode to limit false positives resulting from this.

2.5.4 Workload Selection

All three modules in our study provide functionality related to file I/O. Two of them (`btrfs` and `f2fs`) are file systems and the third one (`nvme`) provides support for an interface standard for storage devices. This allows us to apply the same workload to all three modules. Specifically, the workload consists of the following sequence of steps:

1. Loading the target module and any other required modules;
2. Creating a file system (F2FS for the `f2fs` and `nvme` modules, BTRFS for the `btrfs` module) on either an NVMe (`nvme`, `f2fs`) or SCSI (`btrfs`) device;
3. Mounting that file system;
4. Creating a new file and writing to it;
5. Creating a new directory;
6. Reading the file;
7. Removing the directory;

8. Removing the file;
9. Unmounting the file system;
10. Removing the target module and all other modules that were loaded in the first step.

The instrumentation is active throughout the execution of the workload (that is, the runtime module is loaded prior to the first step and removed after the last step).

This workload exercises most commonly used file system features and, through module insertion, device registration, I/O activity and removal, also exercises the essential functionality of the `nvme` module.

2.5.5 Results

We report on the experimental results obtained with TrEKer and how they answer the research questions posed in Section 2.5.1. The overall result distribution for runs with activated mutation according to the simple detectors discussed in Section 2.5.2 is shown in Figure 2.4.

RQ 1: Soundness of TrEKer

To answer this question, we analyze if there are any spurious indications of error propagation by comparing memory traces of SFI tests for which the injected mutations have not been activated. In mutation-based SFI, there is a risk that the mutated code fraction does not get executed during the test. As the injected faults cannot have an effect on the correct execution of the workload in this case, any differences in the memory traces are false positives since no error propagation can occur in these cases. To reliably identify these tests, we track the execution of mutated code by dedicated log instructions. We then use TrEKer to analyze their memory traces. Any instances of error propagation indicated by TrEKer are false positives.

Figure 2.5 shows the number of trace deviations detected by TrEKer for different numbers of golden runs used as comparison basis for runs with and without mutation activation. The latter case represents false positives, which are indicated by the dashed lines. For all three modules, we observe a false positive rate below 1 %. From Figure 2.5, we see that the number of detected trace deviations does not change beyond 800 golden runs. Consequently, we use this number as a comparison basis in our further experiments to keep the false positive rate in the presented results below 1 % and the comparison stable.

RQ 2: Soundness of SFI tests with TrEKer

Even if TrEKer is a sound detector, it only improves the *soundness of SFI tests* if silent error propagation actually occurs in these tests. To assess if silent error propagation

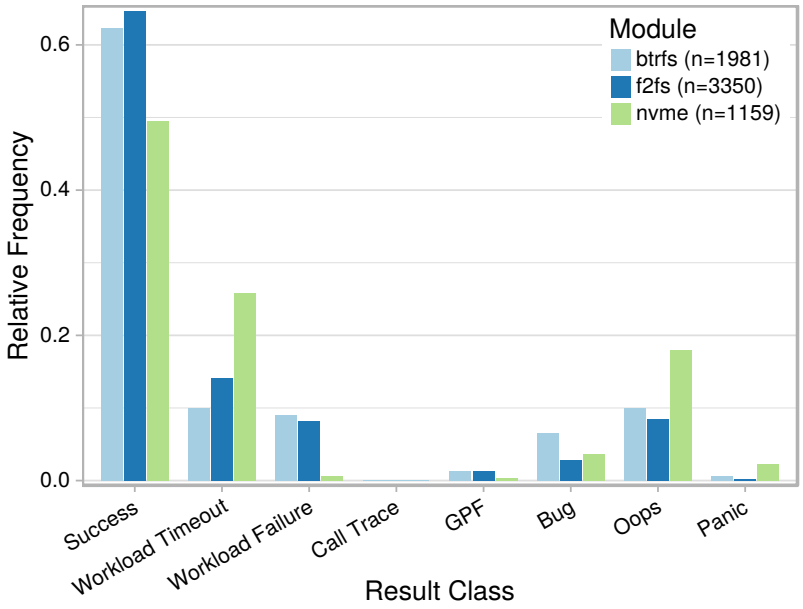


Figure 2.4: Result distribution for runs with activated mutation

occurs and goes unnoticed in conventional OS-level SFI tests, we analyze TREKER's memory traces for SFI tests that appear to complete successfully. To assess the suitability of our approach for detecting divergences in the behavior of faulty versions during such apparently successful runs, we examine the sets of SFI test traces with fault activations which finished without any obvious error indication, i.e. the runs that are marked as successful in Figure 2.4. We show the trace deviations found by TREKER in Figure 2.6 with overall rates ranging from 2.75 % (`btrfs`) to 10.1 % (`nvme`). From the analysis of different visibility types we observe instances of at least two different types of visibility for all modules. However, store visibility via a global variable only occurs for `nvme`. We conclude that, although the different types of visibility occur with different frequencies, analysis of all three is needed to obtain a complete picture of differences in memory access behavior between executions. The significantly higher rate of propagation to the callee rather than the caller is an interesting observation, as it indicates that errors tend to not propagate directly to components that invoke functionality of the targeted modules (i.e., their callers), but rather tend to spread further in the system, at least for the target modules and workload considered in our experiments. While a detailed study is needed to substantiate such a result, this finding illustrates the insights that TREKER fosters and that traditional SFI oracles cannot provide.

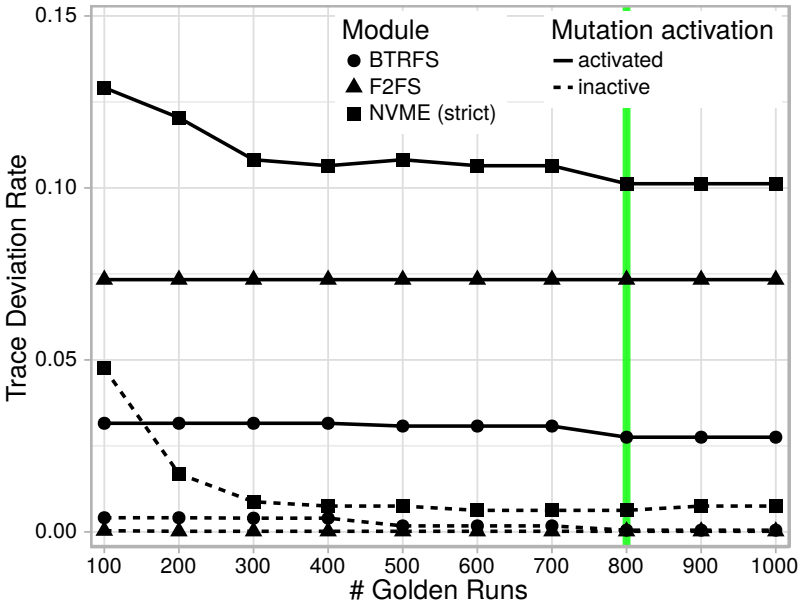


Figure 2.5: Result stability with increasing number of golden runs.

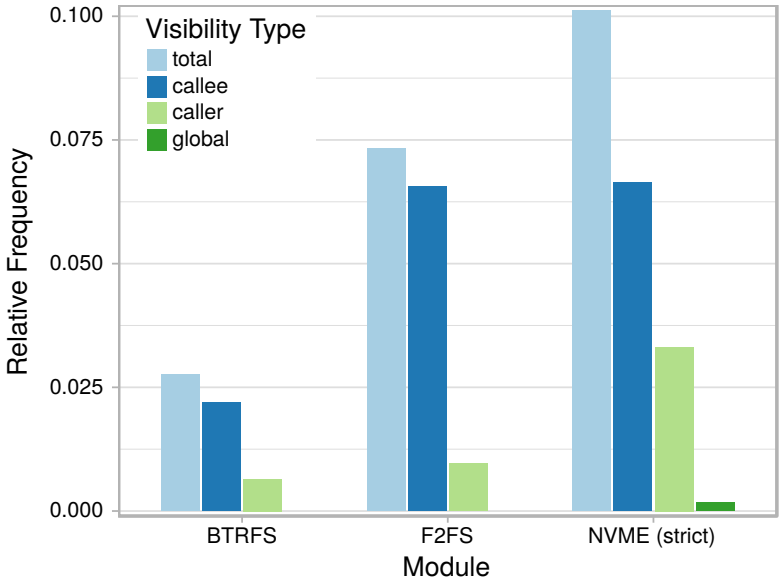


Figure 2.6: Trace deviation rates for the three modules and different types of store visibility when compared with 800 golden runs.

Table 2.1: Compile-time overhead (OH) of instrumentation. User times are reported in seconds.

Module	Build type	Median	MAD	OH
btrfs	instr	71.86	0.21	1.7
	uninstr	43.62	0.11	
f2fs	instr	14.07	0.14	1.4
	uninstr	10.07	0.06	
nvme	instr	2.43	0.02	1.5
	uninstr	1.62	0.02	

RQ 3: Instrumentation Overhead

Static code instrumentation always imposes a certain overhead at both compile-time and run-time. Compile-time overheads are caused by necessary additional code analyses and instruction insertions. Run-time overheads, on the other hand, are caused by the execution of the inserted additional instructions. To assess the overhead associated with our instrumentation, we compare both compilation and execution times of TrEKer against native SFI test compilation and execution in different instrumentation modes.

Compile-time Overhead: Using the GNU `time` utility, we measure the user time that `make` needs for building instrumented and uninstrumented versions of our faulty versions from a clean work space. Table 2.1 summarizes the median and the Median Absolute Deviation (MAD) of user times. Column *OH* reports the overhead factors (between median values) for compilation with instrumentation for all modules. In the median, the compile-time overhead ranges from a factor of 1.4 for `f2fs` to a factor of 1.7 for `btrfs`. We deem these overheads as manageable in practice, especially since compilation is often a one-time effort and the actual needed real-time for compilation is much smaller than accumulated user time due to parallel compilation capabilities of build tools like `make`.

Run-time Overhead: We run the same SFI tests using the full set of faulty versions in three different modes:

1. Without instrumentation,
2. with instrumentation compiled into the faulty versions but disabled during runtime,
3. and with active instrumentation.

We measure the durations of all workload executions that complete in all three modes. Table 2.2 summarizes the median and the MAD of workload durations in

Table 2.2: Run-time overhead (OH) of instrumentation. Workload durations reported in seconds.

Module	Mode	Median	MAD	OH
btrfs	instrumentation active	2.714	0.139	2.5
	instrumentation inactive	1.113	0.012	1.0
	uninstrumented	1.109	0.008	
f2fs	instrumentation active	1.951	0.061	19.3
	instrumentation inactive	0.101	0.006	1.0
	uninstrumented	0.101	0.006	
nvme	instrumentation active	2.133	0.082	3.3
	instrumentation inactive	0.656	0.009	1.0
	uninstrumented	0.642	0.008	

seconds of real time. Column *OH* reports the overhead factors (between median values) compared to the uninstrumented execution. The overhead for runs with active instrumentation ranges from a factor of 2.5 for `btrfs` to a factor of 19.3 for `f2fs`. We attribute the higher relative overhead for `f2fs` to the high concentration of logging output in its mount routine. We expect to achieve a lower overhead for such cases if data logging is changed to use a more efficient format rather than relying on the kernel's `printk` facilities. Execution with inactive instrumentation imposes a negligible overhead. We observe the highest overhead for `nvme` with 14 ms. By comparison, PinOS [BL07] overheads with inactive instrumentation range from a factor of 12 to 120. DECAF [Hen+16] incurs a 15.2 % overhead with disabled instrumentation in addition to the overheads incurred by QEMU emulation. TREKER, in contrast, can run on bare metal configurations to avoid this overhead. We conclude that, with TREKER, instrumented modules could even be used in production, but data logging should be enabled only for tests or execution periods of interest for trace analysis.

RQ 4: INSTRUMENTATION IMPACT ON SFI TEST RESULTS

As code instrumentation modifies the SFI target's binary code and thereby potentially its behavior, it is conceivable that the results of SFI tests are affected or even invalidated by the instrumentation. In order to assess if such an effect is observable for our approach, we compare the results of SFI tests with and without instrumentation of the injection target using Fisher's exact test for independence.

We use the same set of tests with the same three instrumentation modes that we used to assess the run-time overhead in Section 2.5.5. We consider all tests with activated mutation and compare the obtained result distributions for each module. We use Fisher's exact test to test the null hypothesis (H_0) that "there is no association

Table 2.3: *p*-values of Fisher's test of independence of observed result distribution and instrumentation mode

Module	p
btrfs	0.9834
f2fs	0.9978
nvme	0.8420

between observed result distributions and the instrumentation mode". Table 2.3 reports the *p*-values obtained from Fisher's test. With $p \gg 0.05$ for all three modules, we cannot reject the null hypothesis, i.e., there is no statistically significant evidence that the instrumentation systematically changes the result distribution.

Nonetheless, in pairwise comparisons of executions of the same faulty version with different instrumentation modes, we observe a small number of differences in outcomes, which we investigate here. We focus on the comparison between runs with activated instrumentation and uninstrumented runs and see a total of 79, 101 and 68 differences for `btrfs`, `f2fs` and `nvme`, respectively, amounting to 0.79 %, 1.11 % and 2.46 %. As it is the module that most frequently exhibits such divergences, we discuss the nature of the divergences seen for `nvme` in some more detail: In 26 of the 68 cases, we observe a timeout only for the run with activated instrumentation. We hypothesize that these are most likely cases of spurious timeout detection, potentially a result of the overheads we discuss in Section 2.5.5. In a further 21 cases, we observe neither a success nor a timeout but different failure modes. For instance, there are several cases in which the uninstrumented run results in a kernel panic whereas the run with activated instrumentation merely results in a kernel oops before reaching the execution time limit. These are, once again, likely related to longer test execution times due to the instrumentation. We also observe two cases in which the instrumented run completes successfully whereas the uninstrumented run does not, suggesting non-deterministic behavior by the faulty version. Among the remaining 19 cases, we observe twelve in which the instrumented run fails shortly after activating a mutation whereas the uninstrumented run does not (our data does not reveal whether the mutation was activated during the uninstrumented run), six cases in which the instrumented run results in a failure after the end of the workload execution and after removal of the runtime but prior to system shutdown and finally one in which the uninstrumented execution times out but the instrumented run does not.

We conclude that most of the differences in outcome we observe are related to timeout detection and execution time limits and could be tackled by adjusting the corresponding values at the cost of a lower test throughput, similar to what has been reported in [Win+15].

Threats To Validity

We identify the following threats to validity:

1. Non-determinism in the memory access patterns of the target modules that can, even in the absence of faults, lead to divergences between execution traces for the same workload;

2. Limitations of the presented approach for identifying visible stores, assigning symbolic addresses and detecting divergences;

3. The choice of target modules, SUT and workload.

We take several measures to minimize the effects of non-determinism: We use a large number of golden runs as a comparison base, assign symbolic values to memory addresses (including pointers that are used as value rather than address operands in load or store operations) to avoid non-determinism introduced by concrete address values, and handle different processes and threads individually as opposed to explicitly tackling concurrency. The low false positive rates obtained in our evaluation demonstrate the effectiveness of these measures.

TrEKer has several restrictions on the scope within which, for instance, store visibility is determined (e.g., only stores between the prior external function and the current one are considered) or symbolic values are assigned (pointer arithmetic that is not modeled by `getelementptr` instructions is not analyzed). These restrictions result from the deliberately limited scope of our instrumentation and from performance optimizations in the trace processing. Consequently, there may be visible stores outside of the range considered by TrEKer or different memory addresses that are assigned the same symbolic address. Such instances may result in the proposed approach reporting fewer divergences than actually exist.

Finally, the evaluation targets three different kernel modules providing related functionality, running on one kernel version and one system setup. Other categories of kernel modules or other operating systems may behave in a significantly different manner, and our results may not generalize. Furthermore, different workloads could exercise different parts of the module. Long-running workloads, for instance, may be expected to spend less time executing parts of the module for which the instrumentation is particularly expensive, such as module insertion, potentially leading to lower mean overheads. Furthermore, the likelihood of error propagation may increase with longer workload running times. We believe that TrEKer is applicable to a wide variety of usage scenarios and our evaluation demonstrates the viability of the approach.

2.6 Conclusion

In this chapter, we have presented TrEKer, an approach for identifying how faulty OS components can affect other parts of the system, even in the absence of externally visible failures. TrEKer enables tracing memory accesses in a target module using compile-time instrumentation and achieves low instrumentation overheads. We have presented a method for utilizing TrEKer to improve oracles for SFI experiments targeting OS kernel components. An evaluation with several widely used modules for the Linux kernel demonstrates the viability of the approach, finding that conventional oracles would misclassify up to ~10 % of seemingly successful runs. The evaluation shows a false positive rate below 1 %.

3 Accelerating Software Fault Injections

In Chapter 2, we described a method for assessing the effects of faults in kernel modules in a monolithic operating system kernel. This approach allows SFI testing of kernel modules with short workloads as potential instances of state corruption can be detected without waiting for them to result in observable failures. Concerns about SFI test latencies also apply to user space software, where they are driven mainly by the large number of faulty versions that need to be executed due to the increasing complexity of software components and libraries. We therefore need to accelerate SFI experiments. To this end, we propose an approach called FastFI, which speeds up SFI experiments in three different ways:

1. FastFI avoids redundant re-executions of the same code with the same inputs across different faulty versions, thereby reducing the overall amount of work that needs to be performed.
2. To make efficient use of the parallel computational resources available in modern systems, FastFI enables parallelizing SFI experiments by executing multiple faulty versions of the same function in parallel.
3. By creating a single binary containing all faulty versions, FastFI reduces compilation time as common code across different faulty versions is not repeatedly re-compiled.

We demonstrate the applicability of FastFI with SFI experiments on applications from the PARSEC benchmark suite. The contents of this chapter are based on material from [Sch+18a].

3.1 Overview

Software is growing increasingly complex in all parts of the software stack. This includes the OS kernel as well as common user space systems and application software. Coping with the demands on developers that this increase in complexity brings with it is challenging without relying on re-using existing code, frequently in the form of *off-the-shelf* software components. This is cost effective and can benefit system reliability, as such components are likely to be more thoroughly tested

than implementations of the same functionality developed from scratch. However, this kind of software re-use can in some cases also have an adverse effect on system reliability. Such cases arise when the component is used outside of the operational contexts anticipated by its developers. Under conditions outside of its specification, even correct software can malfunction in undesirable ways and adversely affect system reliability. For instance, Leveson and Turner note improper software re-use as a possible contributing factor in the Therac-25 accidents [LT93] in which several people died due to radiation overdoses. Even outside of the realm of safety-critical software, assessing the effects of software faults in parts of the software stack on a software system as a whole is crucial to understanding its overall dependability. To that end, SFI is a widely used technique [CN13; DM06; Voa+97].

As previously outlined in Chapter 1, SFI works by injecting faults into an injection target to create several faulty versions. These faulty versions are then executed with a specific workload, and their behavior is monitored. SFI tools typically inject faults by applying certain transformations to specified code patterns, such as deleting a branch from an `if/else`-statement. These *fault models* [CB89; KIT93b; Nat+13; Rod+99] are closely related to the mutation operators used in mutation testing [Bud+80; DO91; JH11b]. For any such code pattern, a larger code base most likely contains more locations in which it occurs. This means that there are also more locations in which the specified transformations can be applied, which in turn results in a larger amount of faulty versions. Executing more faulty versions naturally requires longer execution times, hence overall SFI test latency increases for larger projects. For complex projects, this can result in large numbers of required experiments as reported in prior studies [Arl+02; Di +12; KD00; Nat+13].

Moreover, some studies investigating *simultaneous fault injection* have found that certain dependability issues can only be revealed by combinations of multiple faults that are injected at the same time [Gun+11; JGS11; Lan+14; Win+13]. Such approaches yield a combinatorial explosion in the number of possible faulty versions and therefore in the number of required experiments. In practice, the number of experiments required for exhaustive SFI testing with simultaneous fault injections can quickly become infeasible.

As testing of complex software systems and testing with simultaneous faults are both desirable, strategies to cope with these complexity issues are required. Two fundamental classes of approaches can be distinguished:

1. *Reducing the number of required experiments*: It is possible to reduce SFI test latencies by executing only a subset of experiments, which can be chosen with random sampling, heuristics or search-based approaches [JGS11; JH08; Nat+13; SAM08]. While the likelihood of such an approach missing a relevant test case is highly dependent on the exact strategy that is chosen, any approach that may miss failing tests is unsound.

2. *Increasing test throughput*: The other class of approaches strives to reduce SFI test latencies by increasing test execution throughput. This can be achieved by accelerating the execution of individual tests, or by taking advantage of modern parallel hardware and parallelizing SFI test execution [Dua+06; Las05; OU10]. The overall effect of parallelization on SFI test latencies is limited by the available parallelism and, like the first class, soundness issues may arise in parallelization [Win+13], not due to leaving out relevant tests but rather due to potential interference between concurrently executing tests.

Combinations of the two classes are, of course, also possible, and likely to be necessary when, for instance, attempting to perform experiments using simultaneous fault injections on a complex injection target.

We present an approach spanning both classes, which we call FastFI. Our approach aims to accelerate SFI experiments for user mode software. FastFI reduces the number of executed faulty versions by not running experiments for faults in functions not exercised by a given workload. For the remaining faulty versions, FastFI speeds up execution by avoiding the re-execution of redundant code that is common to multiple faulty versions, as well as enabling parallelization of SFI experiments.

The rest of this chapter is structured as follows: We first discuss related work in Section 3.2. Next, we present the design and implementation of our approach in Section 3.3, followed by its evaluation in Section 3.4. Section 3.5 contains concluding remarks for this chapter.

3.2 Related Work

In this section, we discuss prior work related to our FastFI approach, starting with work on FI test throughput in Section 3.2.1. We then cover work on test parallelization in Section 3.2.2, and finally work on avoiding redundant code execution in Section 3.2.3.

3.2.1 Improving FI Test Throughput

Parallelizing experiment execution to increase FI experiment throughput has been the subject of several prior studies [Ban+10; BC12; Han+10; Mah+12]. Parallel execution can be achieved using virtual machines [Ban+10; Han+10], which provide strong isolation but also incur substantial overhead, potentially resulting in performance interference [Win+13]. Using virtual machines is particularly costly if setup and teardown are required for each test. A more lightweight isolation can be achieved by using separate processes [BC12]. For FastFI, we choose to rely on processes to avoid the overhead associated with a virtualization-based solution and to take advantage of the process management functionality offered by the OS. Due to

this, FastFI is not applicable to kernel code but rather to software, including systems software, running above the OS layer in user mode.

3.2.2 Test Parallelization

Test throughput is an issue outside of dependability and FI testing, and with the rise of parallel hardware, test parallelization has been studied as a possible solution [Sta00]. Parallelization approaches to improve test throughput have been applied to regression testing [Kap01], testing distributed systems [Las05], and to perform unit testing on cluster hardware [Par+09]. Other recent approaches have investigated cloud-based parallel testing or Testing-as-a-Service, both for test execution [Yu+09; Yu+10] and for static program analysis [CBZ10; Cio+10; Mah+12; SP10].

A common assumption in earlier work on test parallelization is that test cases are generally independent, that is, they do not influence each other and can therefore be executed in parallel or sequentially in an arbitrary order [Dua+06; Mis+07; OU10; Par+09]. More recent work has shown that this assumption is incorrect for several test suites [CMd17; Zha+14]. For this reason, newer approaches have been developed that take potential dependencies between tests into account [Bel+15; Gam+17; LZE15]. Using information about potential dependencies between tests, such approaches can schedule parallel test execution in a manner that avoids altering results compared to sequential execution in the original test suite execution order.

In our approach, FastFI, we handle potential dependencies between SFI tests due to the shared file system at runtime, and otherwise rely on process isolation to avoid interference between test executions. This way, each process has its own address space and file system issues are handled by the runtime, so interferences between test cases can only arise due to either performance interference or due to other, external resources beyond the file system, for example if a test were to communicate over the network. This is a stronger isolation than is commonly assumed for parallel correctness testing, but nonetheless weaker than the isolation achieved by virtual machine-based solutions.

3.2.3 Avoiding Redundant Code Execution

As noted before, FastFI is not purely a parallelization technique. Our approach also increases test throughput by avoiding the repeated, redundant re-execution of code that is shared between several SFI test cases. With VmVm, Bell and Kaiser propose an approach to analyze test suites to determine which data is modified by each individual test, allowing the test suite executor to only reset that part of the system state [BK14]. This way, resets are faster and heavy isolation mechanisms can be avoided. FastFI avoids re-execution of redundant code paths that are shared by many SFI tests and avoids executing faulty versions with faults that the workload would not activate. These ways of reducing execution time are specific to FI testing,

and do not apply to the unit testing scenario targeted by VmVm. With FastFI, we do not attempt to reduce isolation between test cases to improve performance. Rather, we aim to use the most lightweight isolation possible that is still sufficient to safely execute SFI tests in parallel.

3.3 FastFI Approach

In this section, we describe the design and implementation of FastFI, our approach to accelerating SFI test execution. We first give an overview of FastFI and the associated workflow in Section 3.3.1. We then describe the FastFI execution model in Section 3.3.2. Next, we discuss the parallelization strategy and the necessary control logic in Section 3.3.3. The static analysis required for FastFI is covered in Section 3.3.4. We then discuss the technical limitations of our approach in Section 3.3.5. Implementation details of our prototype implementation are described in Section 3.3.6.

3.3.1 Overview

For FastFI, we assume the same basic workflow for SFI tests outlined in Chapter 1 as our starting point: A number of faulty versions of the target are generated, typically containing a single fault each, and compiled. Each faulty version is then executed and their behavior during execution is monitored. This usually involves external experiment control logic that schedules the execution of the faulty versions — in the common case, this can be as straightforward as running each of them sequentially — and monitors the outcome of the SFI tests. In the absence of more sophisticated oracles like TrEKer, this monitoring typically involves checking whether the target crashed, timed out, indicated an error, or finished successfully.

The key difference between this conventional workflow and FastFI is that, with our approach, the generated faulty versions are no longer compiled and executed separately. Rather, we integrate all faulty versions into a single test executable. To do so, we first generate faulty versions as usual. These are then grouped by the functions in which the faults are located. Other groupings, such as at the basic block level, would also be possible, but may limit parallelization opportunities or incur greater runtime overhead. We believe function-level grouping to be an obvious choice for procedural languages and our results suggest that it is effective, as shown in Section 3.4. For each function, each faulty version is extracted and included in the target program, along with an unmodified copy of the original function. The original function is then replaced with FastFI runtime logic. This runtime logic takes over the responsibilities of the experiment control logic in conventional workflows: When the execution reaches a point where a faulty version of a function needs to be chosen, the runtime logic picks a version to execute, forks a new process to execute

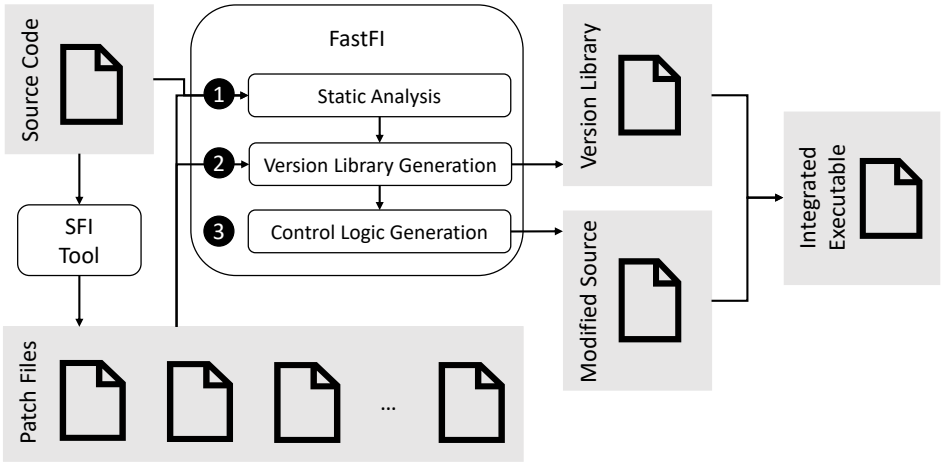


Figure 3.1: Overview of the FastFI workflow.

the faulty version, and monitors execution outcomes. With this approach, no external control logic is required. The integrated executable contains all faulty versions and all necessary control logic to execute and monitor a complete set of SFI tests for a given target.

An overview of the FastFI workflow is shown in Figure 3.1. The original source code is first processed by an SFI tool to generate a set of patches, one for each faulty version. For our experiments, we use the SAFE SFI tool [Nat+13], but FastFI is able to work with any SFI tool that can generate patch files in the unified diff format and only modifies a single function at a time. Both the original source code and the set of patch files are then provided as input to the FastFI tool, which processes them in three steps:

1. *Static Analysis:* In this step, FastFI extracts function location information from the input source code, parses the patch files, checks whether each patch modifies exactly one function, and then performs the mapping of patches to modified functions.

2. *Version Library Generation:* In the second step, FastFI applies each patch to its corresponding function to obtain the resultant faulty version. All faulty versions and a copy of the original version of each function are collected in a library which can then be compiled into the target program.

3. *Control Logic Generation:* In the final step, FastFI replaces each original function for which at least one faulty version exists with runtime control logic.

The result of this process is a modified version of the input source code containing FastFI control logic for all functions for which faulty versions were generated as well

as a library containing all faulty and original versions of the same set of functions. Once the modified source code and the version library are compiled, the result is an integrated FastFI test executable, capable of performing a complete set of SFI tests without external control logic.

3.3.2 FastFI Execution Model

As briefly described above, FastFI merges all faulty versions along with the control logic required for SFI experiments into a single executable. This entails a novel execution model unlike the one used in conventional SFI testing. In this section, we describe and contrast both the conventional execution model and the novel execution model introduced with FastFI.

Conventional Execution

In conventional SFI testing, each faulty version is compiled and executed separately. This execution model is illustrated in Figure 3.2, where an example in which five tests are executed is shown. The figure shows function-level execution traces for each execution. The f_i denote the different functions that are executed. f_i' denotes a faulty version of f_i, and f_i'' another faulty version of the same function. The first trace represents a golden run, an execution of the original software without injected faults. This trace shows functions f_1 through f_5 being invoked in order. The second and third traces represent executions of SFI tests in which a fault was injected in f_5, whereas the last two traces represent executions in which faults were injected in f_4. Each trace can only contain at most one faulty version of one function, although that faulty version may be invoked multiple times. Moreover, a trace containing a faulty version of a function must not also contain the original version of the same function.

In Figure 3.2, all five traces start with the same sequence of functions, f_1 through f_3, up to the point where a faulty version of a function is first invoked. We term this the common execution prefix. If we consider only the first three traces, they share a longer common execution prefix, encompassing f_1 through f_4. Once a faulty version of a function has been executed, the sequence of functions in the execution trace can differ from the golden run and other faulty executions as an activated fault can arbitrarily alter the program state, thereby affecting control flow. For example, in the fourth trace in Figure 3.2, f_5 is not invoked anymore, and f_6 is executed instead. This illustrates that, although multiple SFI tests share a common execution prefix, a common postfix cannot be assumed to exist. Not spending time on repeatedly re-executing common execution prefixes is a core part of the improvements FastFI offers over conventional SFI test execution. We detail how FastFI can avoid these redundant re-executions in the following section.

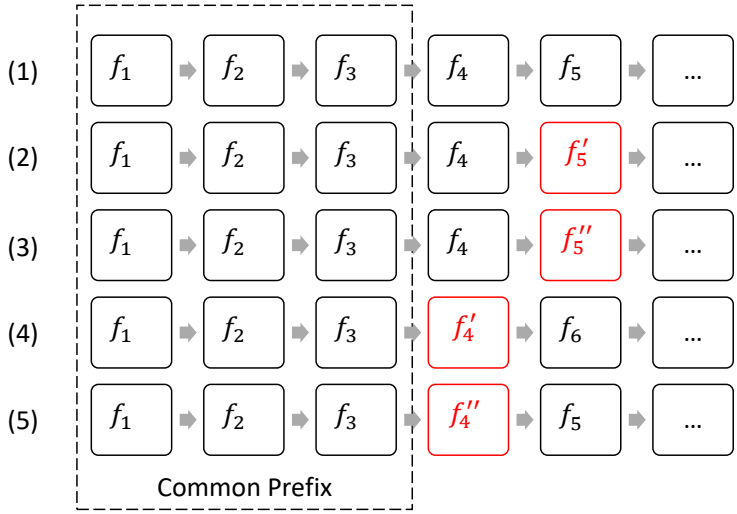


Figure 3.2: Conventional Execution Model. f_i denote functions and f_i' denote faulty versions of a function.

FastFI Execution

What sets FastFI apart from the conventional execution model for SFI tests is that FastFI does not re-execute the common execution prefix for each faulty version. This is possible because FastFI executables contain all faulty versions of the target program. This makes it possible to choose the faulty versions to be executed at runtime, on demand, when a function for which such faulty versions exist is executed for the first time. An example of the FastFI execution model is shown in Figure 3.3. Similarly to Figure 3.2, the illustration shows function-level execution traces. Unlike in the conventional execution model, the common execution prefix consisting of f_1 through f_3 is only executed once. This execution takes place in the master process, which schedules the execution of faulty versions of functions, but never executes such a faulty version itself. When the master process reaches a function for which faulty versions exist for the first time, the FastFI control logic for that function is invoked. This control logic then forks new child processes, shown by the thicker gray arrows in Figure 3.3, each of which executes a single, specific faulty version.

In the example shown in Figure 3.3, the master process executes f_1 through f_3. Once it reaches f_4, a function for which two faulty versions exist, the FastFI control logic is invoked, which forks a new process each for f_4' and f_4''. The fork system call creates an exact copy of the calling process, so the newly created child processes start at the same point in the execution trace at which the master process forked them. They then resume execution, and execute f_4' and f_4'', respectively. As these faulty versions are executed in separate processes, they are isolated from the master process and each other through the process isolation provided by the OS. While

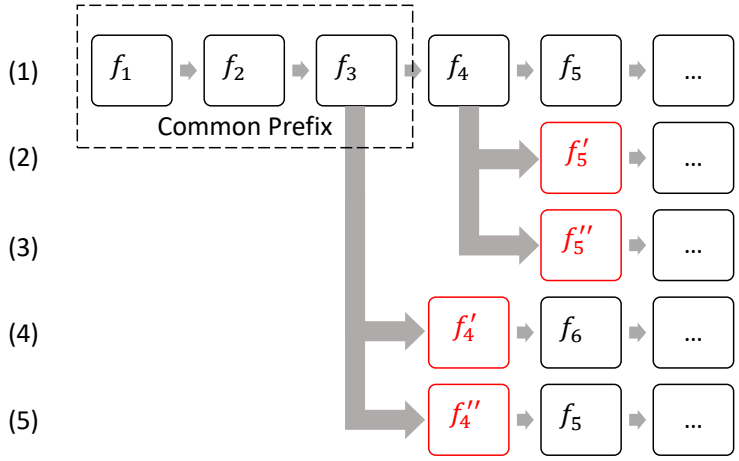


Figure 3.3: FastFI Execution Model. f_i denote functions and f_i' denote faulty versions of a function. The thicker gray arrows represent process forks.

this means that they have completely separate address spaces, it does not prevent interference between processes due to shared external resources. The most common example for this are interferences arising from multiple processes attempting to operate on the same file or files. To mitigate this, the FastFI control logic performs file descriptor manipulation and sets up I/O redirection when a new child process is forked.

Once all faulty versions of f_4 have finished executing, the master process resumes its own execution and invokes the fault-free version of f_4. When it reaches f_5 it once again forks new child processes for each faulty version, waits for them to finish, and then continues executing. This continues until the master process has executed the entire workload. As the master process itself never executes any faulty versions, its own execution trace corresponds to a golden run. Moreover, for every function executed by the master process for which faulty versions exist, all faulty versions have been executed. Therefore, all reachable faulty versions have been executed. Faulty versions of functions not encountered by the master process, on the other hand, have not been executed. This is how FastFI can reduce the amount of faulty versions that need to be tested. Note that this happens entirely dynamically, so a new workload — which may invoke different functions — can be executed without requiring any new analysis or a new integrated executable.

By reducing the amount of faulty versions that are executed and by avoiding the redundant re-execution of common execution prefixes, FastFI can reduce the overall amount of work that needs to be performed to execute a complete set of SFI tests. Additionally, FastFI can further reduce SFI test latencies by taking advantage of modern parallel hardware and executing SFI tests for the same function in parallel. In the example shown in Figure 3.3, f_4' and f_4'' can be executed in parallel, as can f_5'

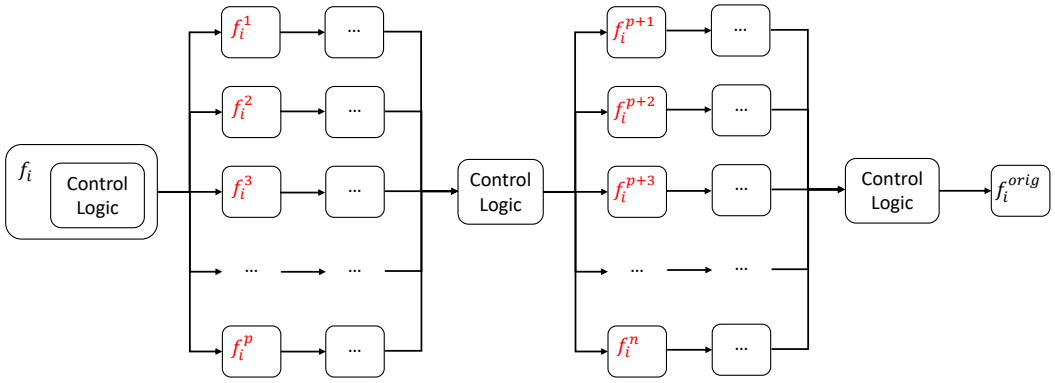


Figure 3.4: FastFI Parallel Execution. The example illustrates the execution of all n versions of function f_i using parallelism degree p in two groups.

and f_5''. Since each faulty version is executed in its own process to ensure address space isolation, applying parallelization is straightforward and does not incur additional overhead. We describe the parallelization strategy used by FastFI in detail in the following section, along with the control logic FastFI utilizes to schedule and monitor the execution of faulty versions.

3.3.3 Scheduling & Monitoring of Faulty Versions

During the generation of a FastFI integrated executable, the function body of every function for which faulty versions exist is replaced with the corresponding FastFI control logic for that function. That control logic is responsible for scheduling faulty versions at a given degree of parallelism as well as monitoring the execution outcomes. A customized version of this control logic is generated for each function.

Parallelization Strategy

To parallelize SFI test execution with a given degree of parallelism p, FastFI dynamically splits all faulty versions of each function in groups of size p. The degree of parallelism is a runtime parameter and can be altered for each execution of the integrated FastFI executable. As illustrated in Figure 3.4, when FastFI execution reaches a function f_i for which n faulty versions exist, the control logic is invoked, which then forks up to p new processes at the same time. Once this group of faulty versions has finished executing, the control logic logs the execution outcomes and forks the next group. This is repeated until all faulty versions of the function have been executed. In the example shown in Figure 3.4, this is the case after the second group has finished executing. At that point, the control logic invokes the original version of f_i, and the master process resumes processing the workload.

The groups of faulty versions that are scheduled for parallel execution are generated by partitioning the set of available faulty versions into subsets of size p. The final subset may be smaller if the total number is not evenly divisible by p. These subsets are then executed one after another. The total execution time for all faulty versions of the function is therefore the sum of the longest execution time of each subset. To keep overall execution time short, and make efficient use of the available computational resources, it is desirable for faulty versions in the same subset to have similar execution times. As execution times are not known ahead of time, they cannot be taken into account when partitioning the set. However, we have made the experience that faulty versions of the same function, generated with the same mutation operator, often exhibit similar execution times. We therefore partition the set of faulty versions based on the mutation operator used to generate them.

Control Logic

As noted above, during the generation of an integrated FastFI executable, the function body of each function for which faulty versions exist is replaced by customized FastFI control logic. A copy of the original function body is included in the version library and invoked by the control logic once all faulty versions have been executed. Original versions are also used by child processes executing faulty versions of other functions as only one faulty version is active in each such process.

We provide a simplified version of the control logic for a function f in Figure 3.5. The control logic starts by first checking whether the FastFI execution model should be used in line 2 or whether the execution of a single, specific faulty version has been requested. In the latter case, that function is called in line 21 and no further actions are performed. This mode exists to facilitate testing the behavior of specific faulty versions under various workloads without requiring separate compilation or recompilation.

The former case, in which the FastFI execution mode is requested, is somewhat more complex. Its implementation can be found in lines 3 to 19. As the FastFI implementation is fork-based, the master process and all forked child processes share the same code. The control logic therefore needs to distinguish in what context it is currently executing.

To that end, it first checks whether it is currently executing in a child process in line 3, and, if so, whether the current function f is the active function (line 4), that is, whether f is the function for which a faulty version should be executed. If it is, the active faulty version is called in line 5. Otherwise, execution resumes with the original version in line 7. In both cases, the control logic passes through the arguments passed to f and returns the return value of the called function.

If, on the other hand, the control logic determines that it is the master process, the desired behavior depends on whether f has already been seen during the current execution. The control logic checks this in line 10. If f is encountered for the

```
1:  function F(args)
2:      if in_fastfi_mode then
3:          if in_child then
4:              if IS_ACTIVE(f) then
5:                  return INVOKE_VERSION(f, args, active_version)
6:              else
7:                  return INVOKE_VERSION(f, args, original_version)
8:              end if
9:          else
10:             if !SEEN(f) then
11:                 subsets ← PARTITION_FAULTY_VERSIONS(f)
12:                 for all subset ∈ subsets do
13:                     FORK_FAULTY(subset, args)
14:                     WAIT_FOR_CHILDREN
15:                 end for
16:                 SET_SEEN(f)
17:             end if
18:             return INVOKE_VERSION(f, args, original_version)
19:         end if
20:     else
21:         return INVOKE_VERSION(f, args, requested_version)
22:     end if
23: end function

24: function FORK_FAULTY(subset, args)
25:     for all faulty_version ∈ subset do
26:         if FORK == 0 then
27:             if FORK == 0 then
28:                 SET_IN_CHILD(true)
29:                 SET_ACTIVE_VERSION(faulty_version)
30:                 return INVOKE_VERSION(f, args, active_version)
31:             else
32:                 MONITOR_EXECUTION(faulty_version)
33:                 EXIT(0)
34:             end if
35:         end if
36:     end for
37: end function
```

Figure 3.5: FastFI Control Logic for a Function f

first time, the control logic proceeds to partition its faulty versions into subsets. It then iterates over those subsets and calls the `fork_faulty` function to execute and monitor the faulty versions, which we will cover in more detail below. The master process waits for all faulty versions to finish executing, notes that the current function has been handled, and then invokes the original version to resume execution in line 18. If f is encountered again later on in the execution, the check in line 10 ensures that the master process will call the original version without forking again.

The implementation of the actual forking logic can be found in lines 24 to 37. Here, the master process iterates over all faulty versions in the current subset and, for each version, calls fork (line 26). Once this is done, the master process returns. The newly forked processes, which we term *monitor processes*, then promptly fork again in line 27 to create the process that will actually execute the faulty version. As shown in lines 28 to 30, the newly created process keeps track of the fact that it is a child process, sets the appropriate faulty version, and then resumes by executing that faulty version with the same arguments originally provided to f and returning its return value. Meanwhile, the monitor process waits for that execution to finish in line 32, where timeouts, error detection, and result logging are also handled. The monitor process then exits.

3.3.4 Static Analysis & Version Library Generation

To insert the control logic in the appropriate places, map faulty versions to functions, and generate a library of faulty and original function versions, FastFI needs information about the static structure of the input source code and the patches generated by the SFI tool. Specifically, for each function, FastFI requires the line numbers where the function is located, the number, names, and types of its arguments, and its return type. For each patch, FastFI needs to know which lines it modifies, which allows FastFI to ensure that the patch only modifies a single function and to map it to that function.

Extracting information from the patches is straightforward as the unified diff format is easy to parse. To obtain the information required from the input source code, we rely on an existing static analysis framework rather than attempting to implement our own parser. Using the information obtained from the input source code and the patch files, FastFI generates a library of faulty and original functions by applying each patch and extracting a copy of the resulting faulty function. The copy is given a unique name and added to the version library. The patch is then reverted so no versions with more than one injected fault are generated and the next patch applied. A copy of the original function is also added to the version library.

3.3.5 Limitations

We discuss the technical limitations of the FastFI approach in this section. FastFI requires the fork system call as specified by POSIX [IEE18], or a way for running processes to clone themselves with similar semantics, to function. The fork system call enables very fast duplication of running processes, but it also imposes some limitations. In particular, multi-threaded programs cannot be cloned with fork as only the calling thread is forked, and the resulting process is only allowed to perform a limited set of operations. For this reason, FastFI is not directly applicable to multi-threaded software which does not include a single-threaded mode.

There are circumstances in which a SUT may behave differently in the FastFI execution mode. This is the case, for instance, if aspects of its behavior depend on process attributes, such as the Process ID (PID), which normally remain constant over the life of a process, but, from the perspective of the SUT, can change in the FastFI execution mode. The same applies to software that makes use of explicit time information as performance in the FastFI execution model may differ, and the master process can remain paused for a significant amount of time if a large number of faulty versions need to be executed. Residual faults in the SUT can also manifest differently between FastFI and the conventional execution model, for example due to differences in the memory layout of the generated executables.

While the process isolation used by FastFI provides isolated address spaces for each faulty version, it does not prevent interferences due to external resources. FastFI takes steps to handle open files but any SUT that interacts with, for instance, network connections or hardware devices would require additional measures to prevent conflicts.

3.3.6 Implementation

We have developed a prototype FastFI implementation that targets software written in the C programming language which execute on POSIX systems. We use the Coccinelle [INR18; Pad+08] for our static analysis as it can be scripted to provide the information FastFI requires about the input source code. Our prototype itself is implemented primarily in Python. As we show in our evaluation in Section 3.4, it is capable of handling real world software.

We note that, while our prototype is currently limited to software written in C, this limitation is not inherent to the approach. With appropriate SFI tools and static analysis frameworks, FastFI could also be applied to, for instance, target programs written in C++ or Rust.

3.4 FastFI Evaluation

To evaluate the applicability and performance of the FastFI approach, we apply our prototype implementation to four real world applications taken from a widely used benchmark suite and written in C. We apply FastFI to these target applications and investigate the following research questions:

RQ 1 Can FastFI reduce test execution latencies for sequential SFI tests?

RQ 2 How do increasing degrees of parallelism affect the execution speedup FastFI can achieve?

RQ 3 Do SFI test results remain stable when using FastFI with increasing degrees of parallelism?

RQ 4 How much can FastFI reduce build times compared to conventional, separate builds?

3.4.1 Experimental Setup

In the following, we describe the execution environment for our experiments, the evaluation targets we chose, and how we conducted our experiments.

Execution Environment

All our experiments are conducted on a system that is equipped with a first generation AMD Ryzen 7 processor with eight physical and 16 logical cores. It is equipped with 32 GiB of main memory and a 1 TiB SSD. The system is running Debian Buster, with a Linux 4.16 kernel provided by the distribution.

Evaluation Targets

As evaluation targets, we choose four applications from the widely used PARSEC benchmark suite 3.0 [Bie11; Pri09]. A brief overview of the chosen target applications is given in Table 3.1. They are representative of different application domains and all of them are written in C, which is a requirement for our current prototype implementation. PARSEC ships with different input sets or workloads as part of the benchmark suite. We use the "simmedium" input sets in our SFI tests. This input set is normally intended for simulations, not native execution, but due to the large number of test cases required for SFI testing, we choose to use it rather than the "native" input set as this allows us to complete our experiments in a reasonable time frame.

Table 3.1: Overview of the four applications from the PARSEC suite used in the evaluation.

Application	Description
`blackscholes`	Numerical financial computations with the Black-Scholes equation
`dedup`	Data stream compression and deduplication
`ferret`	Content-based image similarity search
`x264`	H.264/AVC video stream encoder

Table 3.2: Overview of the number of faulty versions generated by SAFE for each application.

Application	Faulty Versions
`blackscholes`	416
`dedup`	662
`ferret`	6157
`x264`	13368

Experiment Execution

In order to answer the research questions posed above, we take the following steps for each of our evaluation targets:

1. We apply SAFE [Nat+13; Nat13], an SFI tool, to the target software to generate set of mutation patches. Each of the patches generated by SAFE corresponds to a faulty version that needs to be executed during SFI testing. An overview of the number of generated patches for each of our evaluation targets is given in Table 3.2.

2. We perform the static analysis described in Section 3.3.4. To this end, we apply our Coccinelle-based static analysis tool to the evaluation target. We also parse the patches generated in the previous step and perform the mapping between functions and mutation patches.

3. We generate the version library by applying each mutation patch and extracting the modified, faulty version of the function. We also save an unmodified copy of each function to the version library.

4. We replace the original function bodies with FastFI control logic as described in Section 3.3.3.

5. We build an integrated, FastFI-enabled executable using the PARSEC default build configuration "gcc-serial". This results in a single-threaded executable, which is a requirement as outlined in Section 3.3.5.

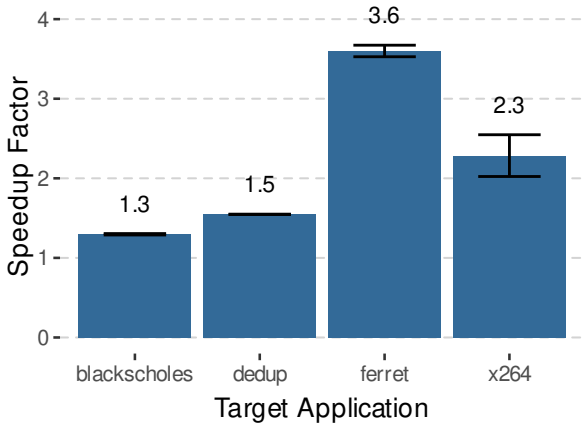


Figure 3.6: FastFI speedup relative to the conventional execution model for sequential execution. Error bars indicate minimum and maximum speedup.

Table 3.3: Percentage of and absolute reduction in the amount of faulty versions executed during sequential FastFI execution.

Application	% of Faulty Versions	Reduction
`blackscholes`	100%	0
`dedup`	75.7%	-161
`ferret`	47.9%	-3206
`x264`	67.8%	-4306

6. We perform SFI experiments using our integrated executables. Each experiment is performed three times. We report averages.

3.4.2 RQ 1: Sequential Speedup

To investigate how FastFI impacts sequential SFI execution latencies, we compare how FastFI performs without parallelization — that is, with a parallelization degree $p = 1$ — to the performance we achieve by executing each faulty version on its own, making use of the ability of integrated FastFI executables to execute a single, specific faulty version as described in Section 3.3.3. In this mode, the faulty version that should be executed is determined prior to execution, most of the FastFI control logic is inactive, and no forking takes place. This means that executions in this mode do not benefit from FastFI's ability to avoid the repeated re-execution of common execution prefixes or its ability to avoid the execution of faulty versions of functions not triggered by the chosen workload. The resulting execution flow corresponds to a conventional SFI test as described in Section 3.3.2.

As shown in Figure 3.6, sequential FastFI execution results in speedup factors from 1.3 to 3.6. We see the highest speedup of 3.6× for `ferret` and the lowest (1.3×) for `blackscholes`. As these are speedups achieved with sequential execution, they are the result of FastFI reducing the total amount of work performed. FastFI achieves this by reducing the amount of faulty versions that need to be executed and by avoiding the redundant re-execution of common execution prefixes. Table 3.3 shows the percentage of faulty versions executed by FastFI relative to the total amount executed in the conventional execution model. We see reductions for three out of the four benchmarks. In those cases, FastFI executes fewer versions than the conventional execution model by skipping unreachable versions automatically. The largest reduction in the number of executed versions is achieved for `ferret`, where FastFI executes fewer than half of all faulty versions. This reduction is also reflected in the fact that `ferret` is the benchmark for which FastFI achieves the highest sequential speedup. `x264` and `dedup` see the second and third largest reduction in the number of executed faulty versions and correspondingly the second and third highest sequential speedup. For `blackscholes`, FastFI executes all faulty versions but still achieves a speedup of 1.3 over conventional execution. This reduction is a result of FastFI's ability to avoid re-executing common execution prefixes.

We conclude that FastFI is able to avoid the execution of faulty versions that are not triggered by the workload and can effectively avoid re-executing common execution prefixes. This allows FastFI to substantially speed up sequential SFI test execution, with a best case speedup of 3.6 in our experiments.

3.4.3 RQ 2: Parallel Speedup

In order to determine how increasing degrees of parallelism affect the speedup FastFI can achieve, we run FastFI-enabled SFI experiments on our four evaluation targets with up to 32 faulty versions in parallel. Changing the degree of parallelism between executions does not require new analysis or compilation as parallel execution is handled entirely by the FastFI control logic code. The speedups achieved by FastFI at various degrees of parallelism relative to the conventional execution model are shown in Figure 3.7. At $p = 1$, the speedups correspond to the sequential speedups discussed in the previous section. With increasing degrees of parallelism, the speedup increases further. At $p = 16$, which corresponds to the number of available logical cores in the machine we use in our experiments, FastFI achieves speedups of 7.6 to 20.6 relative to the conventional execution model. Relative to sequential FastFI execution, the speedups are lower as the lower number of faulty versions executed by FastFI and the avoidance of common prefix re-execution no longer affect the speedup. In this case, the speedups range from 5.0 to 8.9. Going beyond the number of available processor cores with $p = 32$ increases speedups even further to 9.8 to 26.0 compared to the conventional execution model and 6.5 to

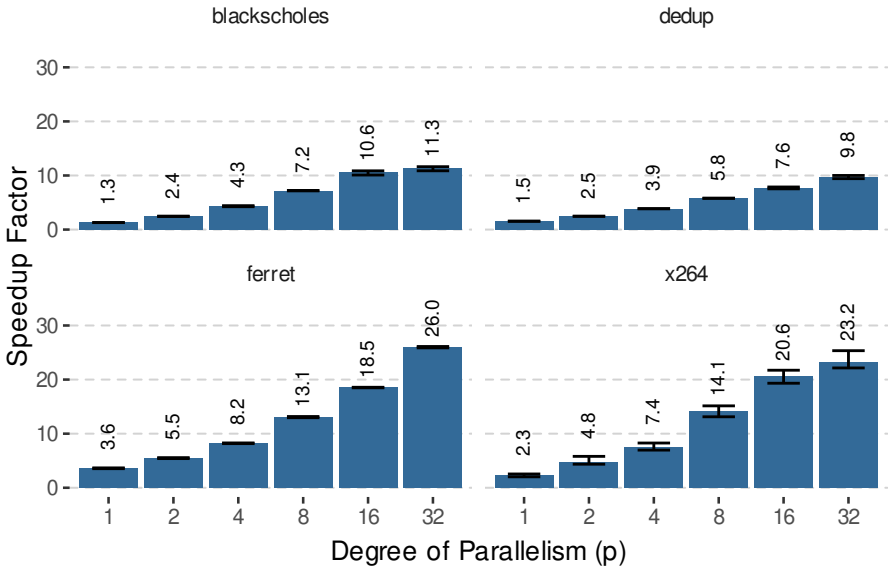


Figure 3.7: FastFI speedup relative to traditional execution model for increasing degrees of parallelization (p). Error bars indicate minimum and maximum speedup.

10.0 compared to sequential FastFI execution. Note that FastFI achieves superlinear speedups compared to the conventional execution model from $p = 1$ to $p = 16$. We believe that further speedups can be achieved by adjusting the FastFI control logic architecture to allow dynamic scheduling of faulty versions as opposed to the static partitioning currently used.

Our results show that parallel FastFI execution can achieve significant speedups over both the conventional execution model and sequential FastFI execution. In our experiments, we achieved speedups of up to 26× for $p = 32$.

3.4.4 RQ 3: SFI Result Stability

As before, we configure FastFI to run at increasing degrees of parallelism from $p = 1$ to $p = 32$. By comparing SFI test outcomes across these different degrees of parallelism, we can assess whether they remain stable or are altered by parallelization. We distinguish between four different SFI test outcomes:

- *Success*: The application finished its execution successfully and without error indications.
- *Crash*: The application crashed. This is detected by checking the status of the child process after its completion.
- *Error*: The application terminated with an error indication. Like *Crash*, this is detected by inspecting the status of the child process.

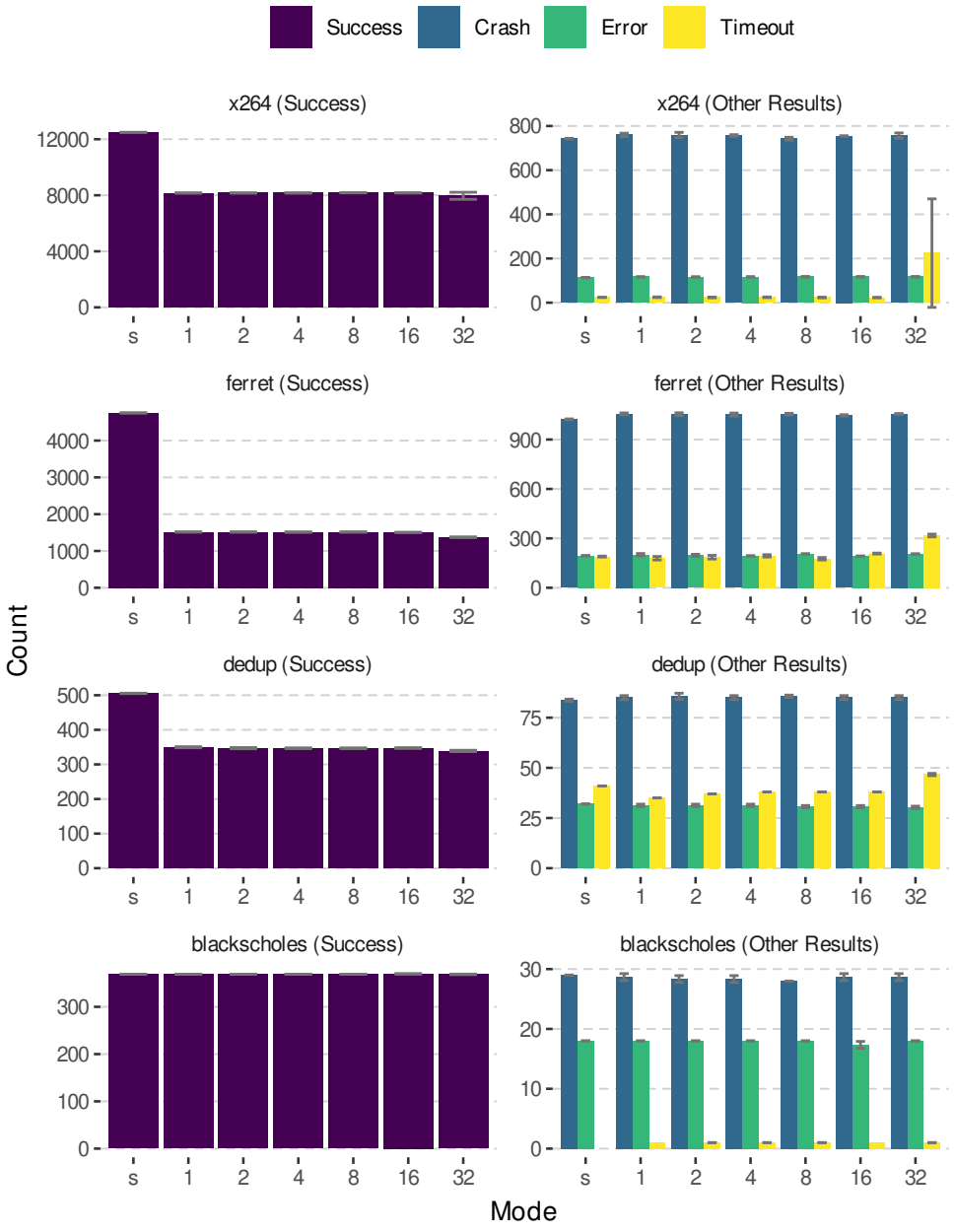


Figure 3.8: SFI test results with FastFI using different degrees of parallelism ($p = 1$ to $p = 32$, on the x axis) and sequential test results in the conventional mode of execution ("s", also on the x axis). Error bars indicate standard deviation.

- *Timeout*: The application did not finish processing the workload in time and was terminated by the monitor process. For each target application, we set the timeout to be 3× the duration of a normal, fault-free execution.

SFI test results for different degrees of parallelism are shown in Figure 3.8. For the sake of readability, we show successful and non-successful runs in separate plots. The columns labeled with "s" show the results of sequential, conventional SFI test executions.

Looking at the successful runs on the left, it is apparent that the number of successful runs in the conventional execution mode is higher than for FastFI with any degree of parallelism for all benchmarks except `blackscholes`. This is due to the fact that the conventional execution model performs a number of test executions for faulty versions in which the fault never gets activated, which naturally results in a successful test completion. FastFI does not execute such versions and therefore performs fewer successful runs. This reduction in executed versions is shown in Table 3.3.

For the other result classes, shown on the right, the results are stable up to $p = 16$ for all four target benchmarks. At $p = 32$, the number of timeouts rises for `x264`, `ferret`, and `dedup`, and for `x264`, the number of timeouts varies substantially across test runs. The number of crashes and errors, however, remain stable. For `blackscholes`, all results remain stable at $p = 32$. As results remain stable up to $p = 16$, which corresponds to the number of available logical cores on our experiment setup, and the only effect at $p = 32$ is an increase in spurious timeouts which could be mitigated by increasing the timeout duration, these results show that FastFI parallelization does not adversely affect result stability.

We conclude that, with appropriate degrees of parallelism and sufficiently long timeouts, using FastFI parallel execution can reduce SFI test latencies without affecting result stability.

3.4.5 RQ 4: Build Time Speedup

To determine whether FastFI is capable of reducing build times in addition to execution latencies, we build an identical set of faulty versions once as an integrated FastFI executable and once by building a separate executable for each faulty version. We use incremental compilation when building the separate executables so that we don't give an advantage to FastFI by re-compiling the entire target benchmark for every faulty version. Instead, only one compilation unit is re-compiled and the final executable is linked again. This is a common approach for building separate binaries in SFI testing. For both modes, the recorded time includes the time required to apply and revert the mutation patches. For FastFI, the recorded time also includes the generation of the version library and control logic.

Table 3.4: User build times for FastFI and separate executables.

Application	Build Time (s)		FastFI	
	FastFI	Separate	Speedup	Reduction
`blackscholes`	2.91	26.11	8.97	23.2 s
`dedup`	18.47	88.96	4.82	1 min 10 s
`ferret`	57.94	809.10	13.96	12 min 31 s
`x264`	2062.90	21508.37	10.43	5 h 24 min 5 s

We report the measured user times in Table 3.4, along with the speedups achieved by FastFI and the absolute reduction in user build time. FastFI achieves substantial speedups over the conventional approach of building separate executables, ranging from 4.82 to 13.96. Moreover, the two benchmarks with the longest build times, `x264` and `ferret`, also see the greatest benefit, both in terms of speedup and in terms of time reduction. For `x264`, the reduction in build time amounts to more than 5 h. The reason integrated FastFI executables can be built more quickly than separate executables is, as with the sequential execution speedups, that FastFI reduces the total amount of redundant work that needs to be completed. In the conventional approach, an entire compilation unit needs to be re-compiled for every faulty version and a new executable needs to be linked, even though just a single function is different from the fault-free version. FastFI avoids this overhead since the approach works at function granularity.

Our results show that integrated FastFI executables can be built significantly more quickly than conventional, separate executables containing a single version each. We achieve speedups of up to 13.96×, and build time reductions of up to 5 h 24 min.

3.4.6 Discussion

Our experimental evaluation of our prototype implementation shows that FastFI is applicable to real world software. It can achieve sequential speedups of up to 3.6, showing that it is effective at avoiding redundant code (re-)execution, both due to inactive faults and common execution prefixes. Moreover, FastFI enables straightforward parallelization, which can yield further speedups. In our experiments, we achieve speedups of up to 20.6 at $p = 16$ without adversely affecting result stability. FastFI can also reduce compilation times substantially due to its function-level granularity, achieving build time speedups of up to 13.96 in our experiments. We conclude that FastFI is a viable approach for reducing both the compilation and execution times required for SFI tests of real world software.

3.5 Conclusion

The increasing complexity of software at all levels of the stack entail an increasing number of SFI experiments that are required for assessing its dependability. With FastFI, we have developed an approach to accelerate SFI tests of user mode software by combining several techniques:

1. FastFI avoids redundant re-executions of common execution prefixes and executions of faulty versions that the workload does not trigger.
2. FastFI enables parallelizing SFI experiments by executing multiple faulty versions of the same function in parallel, thereby taking advantage of modern parallel hardware.
3. FastFI reduces compilation time as common code across different faulty versions is not repeatedly re-compiled.

Our evaluation on four applications from the PARSEC benchmark suite shows that FastFI is applicable to real world software across different application domains. It achieves both sequential and parallel execution speedups, substantially reduces build times, and, given appropriate settings, does not affect result stability.

4 Accelerating Kernel Error Propagation Analysis

Assessing operating system dependability remains a challenging problem, particularly in monolithic systems. Component interfaces are not well-defined and boundaries are not enforced at runtime. This allows faults in individual components to arbitrarily affect other parts of the system. SFI can be used to experimentally assess the resilience of such systems in the presence of faulty components. However, as discussed in Chapter 2, applying SFI to such systems raises its own set of challenges, such as the difficulty of detecting corruptions in the internal state of the system. Although we have developed an approach to tackle this challenge, it requires additional instrumentation which further increases SFI test latencies. In Chapter 3, we presented an approach for accelerating SFI experiments on user space software, which did not consider EPA instrumentation and was not directly applicable to kernel code. In this chapter, we present an approach that leverages static and dynamic analysis alongside modern OS and VMM features to reduce SFI test latencies for OS kernel components in the presence of the instrumentation required for trace-based EPA to enable efficient and precise detection of internal state corruptions. We demonstrate the feasibility of our approach by applying it to seven widely used Linux file systems. The contents of this chapter are, in parts verbatim, based on material from [CSS20].

4.1 Overview

Monolithic OS kernels are large and complex software systems. While they do not enforce boundaries or isolation between components at runtime, they are nonetheless commonly developed as collections of separate modules which may differ in size and complexity and implement different kinds of functionality. Moreover, modules, such as device drivers or file systems, may be developed and maintained by separate teams of software developers, resulting in substantial differences in code quality and, consequently, the number and density of faults in different modules. In order to assess the dependability of an operating system kernel in the presence of faulty modules, we need to analyze how such faulty modules can affect other modules or the core system. A well-established technique for this purpose is SFI, in which faults are deliberately injected into a module as described in Section 1.2. In

this way, a number of faulty versions of a module can be created, and the system can be exposed to the potentially faulty behavior of the module to test how resilient it is to such faults. To achieve this, normally, each faulty version is loaded, the system is subjected to a test workload and the behavior of the system is monitored. If the system fails, reports an error or its behavior otherwise deviates from a normal execution, the faulty module has affected overall system behavior. It is, however, more challenging to determine the test outcome if no such observable deviation has occurred. This may be the case if the injected fault has not been activated by the test workload, or does not affect system behavior or state under the specific workload. However, it may also be the case if the fault is activated and adversely affects or corrupts the state of the system but the SFI experiment ends prior to the effects of that state corruption becoming observable. In keeping with the Laprie taxonomy [Avi+04], we refer to instances in which faults in a module or component affect other parts of the system as *error propagation*. Identifying instances of potential error propagation requires EPA and is particularly important in the context of operating system kernels, as these are typically long-running systems, and the limited test durations typically used in SFI testing do not — and, due to the impact on test latency, cannot — reflect that aspect of a realistic operating environment.

As described in Chapter 2, one way to tackle this issue is by introducing additional instrumentation to the faulty module and using that instrumentation to trace the modifications that the faulty module makes either to the system state or to externally visible parts of its own state. These modifications can then be compared to those made by the non-faulty version of the same module, with the underlying assumption that instances in which the behavior of the faulty version diverges from the non-faulty version constitute potential error propagation. While this yields promising results, it also introduces additional overhead, thereby further increasing the already substantial SFI test latency, especially when instrumenting more complex modules or executing longer workloads. Moreover, false positives are an issue with this kind of detector since the system may exhibit non-deterministic behavior.

We investigate ways of mitigating the impact such instrumentation has on SFI test latency. Long test latencies are a known problem with SFI tests and various means of addressing them have been proposed. We give a brief overview of relevant work in Section 4.2.1. Our proposed approach is conceptually related to one such technique in particular, FastFI, which we describe in Chapter 3.

The rest of this chapter is structured as follows: First, we cover related work in Section 4.2. We then give a more detailed description of our proposed approach and prototype implementation in Section 4.3. Next, we discuss the experimental evaluation of our approach in Section 4.4. We discuss our results as well as potential threats to validity in Section 4.5. Finally, we provide concluding remarks in Section 4.6.

4.2 Related Work

The goal of our approach is to reduce SFI test latencies for kernel code to improve the practical applicability of error propagation analysis, which is often hindered by long test latencies. Related work includes approaches for reducing fault injection test latencies, which we cover in Section 4.2.1), more general work on test parallelization that needs to tackle several related issues, discussed in Section 4.2.2, as well as work on error propagation analysis, covered in Section 4.2.3.

4.2.1 SFI Test Latencies

Test latencies are a well known problem in fault injection and numerous approaches to reduce them have been proposed. Most of these focus on parallelizing the execution of fault injection experiments using different isolation mechanisms to prevent concurrent executions from interfering with one another.

D-Cloud [Ban+10; Han+10] is a cloud system that enables fault injection testing of distributed systems using virtual machines to isolate the systems under test. We also use virtual machines to isolate different SFI experiments from each other but our work targets kernel code rather than distributed systems.

Other techniques rely on more lightweight process isolation to avoid the overhead associated with the strong virtual machine isolation. This includes AFEX [BC12] as well as FastFI, which we describe in Chapter 3. FastFI accelerates SFI experiments by avoiding redundant executions of code that is common to multiple faults, avoiding executions of faulty versions in which the fault location is never reached under a given workload, and facilitating process-level parallelization of SFI experiments. Unlike FastFI, we make use of the stronger isolation guarantees provided by virtual machines, which allows us to apply our technique to more targets, including kernel code, and lets us avoid the handling of potential resource conflicts on file descriptors that FastFI requires.

While virtual machines provide strong isolation, executing multiple fault injection experiments in parallel can nonetheless affect results due to the impact of a higher system load on test latencies, as shown in [Win+15]. In our evaluation, we take care to choose appropriate timeout values for our workload to avoid such effects and compare results across different degrees of parallelism on the same hardware.

4.2.2 Test Acceleration

Outside of the specific context of fault injection testing, numerous ways of speeding up software testing in general have been investigated.

Many such approaches use parallel test execution and rely on the assumption that test cases are independent and can therefore be executed in any order or concur-

rently without altering results (e.g., [Dua+06; Mis+07; Par+09]), but recent work has shown that this assumption frequently does not hold in practice [LZE15; Sch+19]. A related problem arises in fault injection tests, where executing tests in parallel may also lead to interferences. Recent work investigating the feasibility of statically detecting conflicts between software tests [Sch+19] is not applicable to fault injection testing since it relies on each test having a distinct entry point whereas, in a typical fault injection scenario, all tests use the same workload and hence the same entry point.

FastFI, as discussed in Section 4.2.1, relies on address space isolation between user mode processes and runtime handling of file descriptors and does not attempt to tackle issues arising from multi-threaded targets or conflicts on other resources. As we are targeting kernel code, this is isolation mechanism is not applicable and we instead make use of the stronger isolation guarantees provided by virtual machines.

Besides parallelization, test execution can also be accelerated by avoiding redundantly executing code multiple times and not executing unnecessary code. For instance, VmVm [BK14] avoids unnecessarily resetting the entire system state between test executions by determining which parts of the system state each test affects, and O!Snap [Gam+17] leverages disk snapshots to reduce test setup and execution times as well as cost in a cloud setting. Both FastFI and our work avoid repeatedly re-executing the entire workload by cloning system state at appropriate points, but do so using fundamentally different methods: While FastFI relies on forking a process, we clone VM instances as described in Section 4.3.3. This VM cloning resembles the VM fork primitive described in SnowFlock [Lag+09], however, our implementation does not require modifications to the VMM and is intended for cloning VM instances on a single host machine rather than in the distributed or cloud scenarios targeted by SnowFlock.

4.2.3 Error Propagation Analysis

Our work builds directly on TrEKer, our technique for tracing error propagation in OS kernels by using execution traces at the granularity of individual memory accesses as described in Chapter 2. We briefly summarize how TrEKer works in Section 4.3.

Execution traces are commonly used to assess the outcome of fault injection tests [APB14; TP13]. In this context, the execution of an unmodified system is traced one or more times. These executions, called *golden runs*, are then used as an oracle that executions in which faults have been injected can be compared against.

Execution traces can be collected in several ways. Some approaches rely on debuggers [Aid+01; CMS98], while others use full-system simulation [SPS09]. While these techniques allow for fine-grained tracing and control over the system under test, they also induce substantial execution overhead, particularly in the case of full

system simulation. If this overhead is not acceptable, DBI can be used to collect traces (e.g., [Lan+14; ZKB13]). However, DBI is not straightforward to apply to kernel code, and while several tools and frameworks exist [BL07; FBG12; Hen+14; Hen+16; KB13], they have not been used to collect execution traces for fault injection tests. We follow the approach used in TrEKer, which relies on compile-time instrumentation.

4.3 Approach

We propose an approach for efficiently and precisely identifying ways in which faults in a component of a monolithic operating system affect other parts of that system. We first provide a brief overview of the systems we are focusing on and the ways in which their components interact in Section 4.3.1. Then, we explain how error propagation analysis can be conducted in such a system in Section 4.3.2 and how we aim to address the limitations of prior work. This leads to an overview of our proposed approach in Section 4.3.3. We describe our implementation in Section 4.3.4.

4.3.1 System Model

As in our work on OS kernel error propagation analysis described in Chapter 2, we follow the Laprie taxonomy [Avi+04] which we outlined in Section 1.2. We provide a brief recap here. We assume a component-based system in which each component implements a function according to some specification, realized through the externally observable part of its state, or its *external state*. Should that external state deviate from its specification, a component *failure* has occurred, which may in turn have been caused by a preceding deviation (*error*) in the part of the component state that is not externally observable. The cause of such an error is called a *fault*, and the process by which a fault causes an error is called *fault activation*. Alterations in subsequent system states caused by an error are instances of *error propagation*.

As we focus specifically on monolithic OS kernels in our work, these abstract notions map onto the systems we are studying as follows:

- The system is the OS kernel, and system components are individual kernel modules. Kernel modules are conceptually separate entities implementing distinct functionality, but no runtime isolation is enforced. We do not assume the existence of an explicit specification for the function they are intended to implement.

- A component's external state or interface consists of data passed between the component and the rest of the system either through function call arguments

and return values (in either direction) as well as shared memory communication. Since all components share the same address space, all memory accesses constitute potential shared memory communication. We consider externally visible write accesses, as defined in Chapter 2, to be instances of shared memory communication and use an augmented version of the TrEKer analysis to identify them.

- We are focusing on permanent faults in the form of software bugs. We do not consider transient or hardware faults.

4.3.2 Kernel Error Propagation Analysis

As noted in Section 4.3.1, we use a similar system model as in Chapter 2 and base the error propagation analyses aspects of our approach on the TrEKer analysis proposed there. However, the TrEKer approach suffers from certain limitations that we address in this work, most notably the overhead caused by the required instrumentation, which further exacerbates the problem of long test latencies in SFI testing. Moreover, TrEKer requires a large number of golden runs to obtain stable results, which in turn results in increased execution times to obtain these golden runs and long analysis times to process them. Although TrEKer achieves a false positive rate below 1 % with a sufficiently large number of golden runs, given the large number of faulty versions that SFI in complex kernel modules may yield, this can still amount to hundreds of misclassified executions.

We propose an approach to reduce overall SFI test latency, while also requiring fewer golden runs. Our approach is based on integrating all faulty versions of a module into a single binary and leveraging fast VM cloning to avoid redundant executions of common execution prefixes for different faulty versions, thereby also reducing non-determinism between executions. We detail our approach in Section 4.3.3.

4.3.3 Improving Kernel EPA with Fast VM Cloning

As mentioned in Section 4.1, our approach for reducing kernel SFI test latency is conceptually related to that of FastFI, described in Chapter 3, which we briefly recap here. We first summarize the execution model of FastFI in Section 4.3.3 and discuss why it is not directly applicable to OS kernels, which are the target systems we focus on in this work. We then describe our modified approach and its integration with error propagation analysis in Section 4.3.3.

FastFI Summarized

FastFI can reduce improve SFI test latencies in three ways:

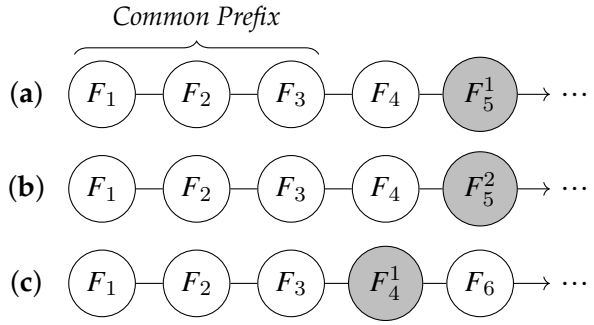


Figure 4.1: The *common prefix problem*. Each F_i is a function and F_i^j is the j$^{\text{th}}$ faulty version of that function after fault injection.

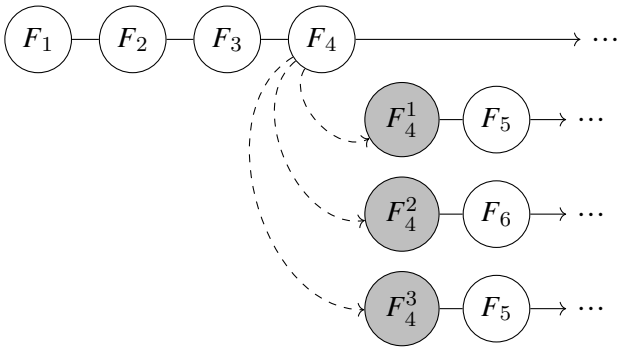


Figure 4.2: The FastFI approach. The redundant re-execution of functions F_i in the common prefix is avoided. FastFI executes each faulty version in a new process once the injected function $F : i^j$ is reached.

1. FastFI avoids repeated executions of code paths which are shared by different SFI tests and therefore faulty versions.
2. FastFI reduces the amount of executed faulty versions by automatically avoiding the execution of faulty versions for which the given workload does not activate the fault.
3. By facilitating parallelization, FastFI can take advantage of modern, parallel hardware to accelerate SFI tests.

latencies by avoiding redundant, It is based on the insight that, in SFI testing, the same workload is executed for each version, and prior to the execution reaching the part of the code that a fault has been injected in, the same code is executed for each version. We refer to this as the *common prefix* problem and show a simplified example in Figure 4.1. In this example, three SFI test executions (a) – (c) are depicted as function-level execution traces. Faulty versions of a function F_i are shown as F_i^j. The code in functions F_1 through F_3 is executed for each test even though it contains no injected faults. The repeated execution is therefore redundant.

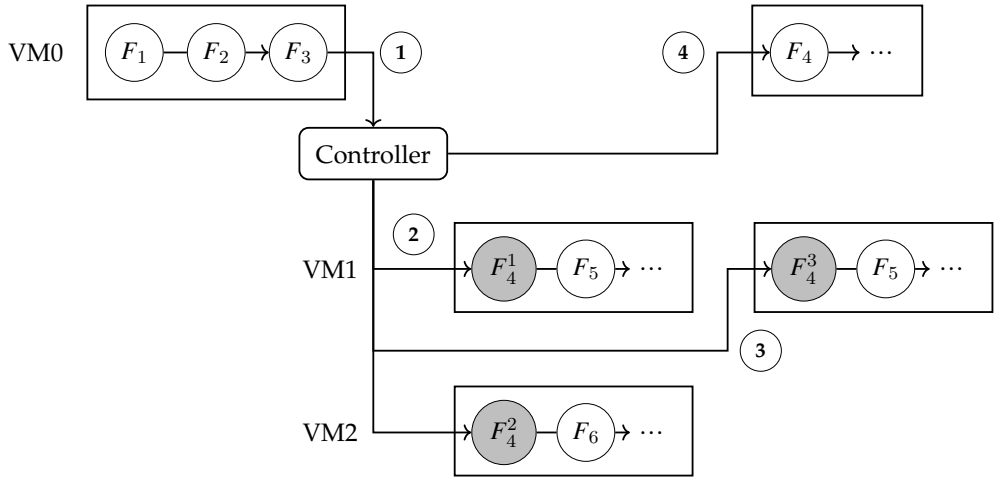


Figure 4.3: Our enhanced approach. Each faulty version is executed in a separate VM, which is started from a snapshot.

The solution proposed in FastFI is shown in Figure 4.2. Here, F_1 through F_3 are only executed once and new processes are forked for each faulty version when a function for which such versions exist is reached. While this approach can achieve substantial speedups as described in Chapter 3, it is not applicable to kernel code for several reasons.

First, FastFI uses forking to duplicate the state of the system under test at appropriate points during the execution. As we are studying kernel code, we cannot rely on abstractions provided by the OS, such as processes. Due to this reliance on the process abstraction, FastFI cannot handle multi-threaded code, the SUT must consist of a single process, and it must not rely on external resources besides the file system. While these are not insurmountable challenges for the systems FastFI has been applied to, they render it inherently unsuitable for kernel code.

Secondly, FastFI integrates the entire control logic — which performs the forking, monitoring and logging — in the SUT, which is not a desirable solution for kernel code as it could impose substantial delays in timing-critical portions of the system and impose additional technical challenges due to the lack of the availability of a process abstraction.

Next, we describe how we address these limitations and how we integrate our enhanced approach with error propagation analysis.

Fast VM Cloning

To overcome the limitations that render FastFI unsuitable for our target systems, we propose a conceptually related approach that makes use of virtual machine snapshots rather than processes, reduces the amount of control logic embedded in the

system under test, and facilitates straightforward integration with trace-based error propagation analysis.

The resulting approach is illustrated in Figure 4.3, which continues our earlier example from Figure 4.2. The execution of the target system starts in a virtual machine, VM0, where, in keeping with the prior example, functions F_1 through F_3 are executed. When function F_4 is reached, the system notifies the controller (①), a separate process running outside the VM. The controller process is where most of the control logic, which was embedded in the SUT in the FastFI model, resides in our approach. At this point, the controller suspends the execution of VM0 and instructs the VMM (we use QEMU [Bel17]) to take a snapshot of the system. The controller then decides how many VMs to create based on the number of available faulty versions of the current function and the desired degree of parallelism. It then spawns the new VMs and resumes execution from the snapshot (②). In this example, two parallel instances are used, VM1 and VM2, and they start executing F_4^1 and F_4^2. Execution is monitored by the controller, and once a VM finishes executing the workload, it is once again suspended, the snapshot is loaded and the next faulty version is executed. In the given example, VM1 finishes executing F_4^1, is restored to the snapshot by the controller (③) and then executes F_4^3. One key advantage our approach of not embedding control logic in the SUT offers is that we can resume execution in VM0 even before all faulty versions have finished executing. Therefore, in the given example, VM0 can resume executing the unmodified system as soon as VM2 finishes and a CPU core becomes available (④). This is particularly valuable in cases where a faulty version causes the system under test to hang until a timeout detector is triggered as such cases no longer completely halt experiment progress, giving our enhanced approach a performance advantage over the FastFI approach. In addition to cloning virtual machines and scheduling the execution of faulty versions, the controller is also responsible for monitoring and logging experiment outcomes. By the time VM0 and all other VM instances that have been spawned have finished, every faulty version that is reachable by the given workload will have been executed.

Our approach also allows for the integration of TrEKer-style EPA. We show this by integrating the existing TrEKer implementation with our new approach by adjusting the TrEKer runtime to send trace entries to our controller and enhancing the TrEKer trace processing to support the trace fragments generated in our modified execution model. This integration offers benefits besides performance improvements which are particularly important due to the overhead incurred by memory access tracing. In particular, since our approach avoids re-executing common prefixes for different faulty versions as well as the original version of the system, the execution trace fragments that have to be stored for later offline analysis are smaller and there is no opportunity for traces to diverge prior to reaching the function containing the fault due to non-deterministic behavior of the SUT. We study the impact

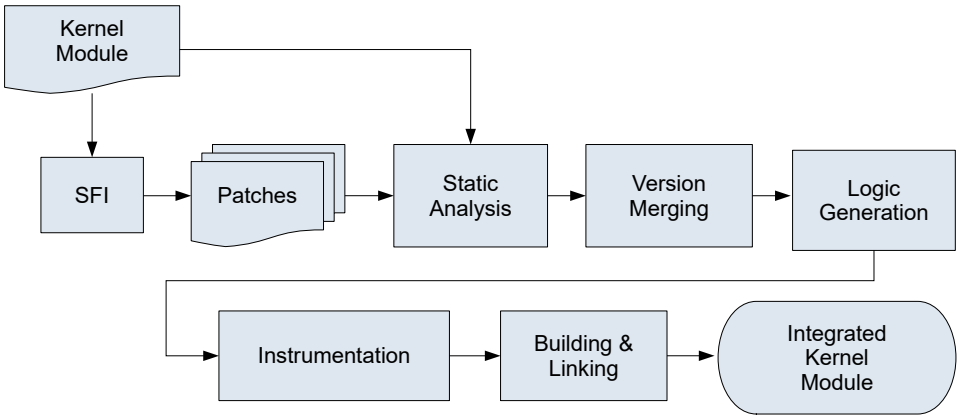


Figure 4.4: An overview of our implementation.

this has on the detection and false positive rates of the error propagation analysis in Section 4.4.

4.3.4 Implementation

Our implementation consists of two parts. The first consists of the analyses and tools required to obtain a single, integrated kernel module containing multiple faulty versions. An overview of this process is shown in Figure 4.4. The second part is the runtime logic, consisting of the controller and runtime kernel module.

Compile Time

We start by applying an SFI tool, SAFE [Nat+13; Nat13], to the target kernel module, resulting in a number of patch files, each corresponding to a single faulty version. Next, we need to determine for each such patch what function it modifies and where that function is located in the original, unpreprocessed source code. We use a custom GCC plugin to efficiently obtain function location information and parse the patch files to create a mapping. With this information, we can then perform the version merging and logic generation steps as shown in Figure 4.4. First, instead of creating a copy of the kernel module for each faulty version, we merge all faulty versions into a single module by creating a copy of the modified function for each such version. We also include a separate copy of the original function implementation. Then, in the logic generation step, we replace the original function implementation with a small amount of runtime logic to determine which version should be executed. Unlike FastFI, this does not include any logic for orchestrating or monitoring the execution of different faulty versions. Rather, our support logic just calls out to a runtime support module, which in turn communicates with the

controller as shown in Figure 4.3 as required. We then instrument, build, and link the resulting code. By only instrumenting after merging all faulty versions, we can avoid redundant instrumentation overhead resulting from instrumenting the same code multiple times. This yields a single, integrated kernel module which is fully instrumented and contains all faulty versions.

Runtime

The runtime part of our implementation consists of two main parts: The first is the controller as shown in Figure 4.3, which runs on the host and is responsible for spawning VMs, controlling parallelism, monitoring experiment outcomes, and logging tracing information. The second is the runtime module, which is implemented as a kernel module that is loaded in the experiment VMs and enables communication between the controller and the experiment VMs as well as providing logging interfaces to the instrumented kernel modules and providing supporting functionality to the version select logic described above.

We implemented the controller as a Rust program which manages the VMs for SFI experiments and performs experiment result detection and logging. While one goal of our implementation is to achieve fast VM cloning, which is a crucial factor in realizing the speedups we are aiming for, we choose not to use a custom VMM to keep the applicability of our approach as broad as possible. Instead, we use QEMU and rely on its existing snapshotting functionality, along with a file system providing CoW functionality on the host, to quickly clone VMs. This lets our implementation take advantage of the mature implementation, numerous features and supported emulated devices of QEMU while still achieving high performance when cloning VMs.

The runtime module is a Linux kernel module written in C. It provides an interface to the integrated kernel module to aid in version selection and handles communication with the controller over a VirtIO serial device. It also passes the log messages required for trace-based EPA to the controller over the same interface. Using a VirtIO device for logging, rather than, for instance, SSH, as in Chapter 2, lets us keep overhead as well as noise on the system to a minimum.

At runtime, when the original VM (VM0 in Figure 4.3) reaches a function for which faulty versions exist, the following sequence of events happens:

1. The version selection logic in the integrated module calls the runtime module and provides the number of faulty versions that need to be executed.

2. The runtime module notifies the controller and blocks until a response is received.

3. The controller stops VM0 and takes a snapshot.

4. Depending on the desired degree of parallelism P, the controller creates up to P copies of the file containing the VM snapshot. To ensure that this step is performant, it is necessary that the snapshots are kept on a file system supporting lightweight CoW copies. In our implementation, we use a RAM-backed XFS file system.

5. The controller creates a workqueue for the faulty versions of the current function and spawns controller threads for each new VM, which in turn start QEMU processes, load the snapshots, send a response to the runtime module indicating which faulty version to execute, and monitor the resulting execution to determine the SFI experiment outcome. These threads also log the tracing data relayed to them by the runtime module.

6. Depending on P, the controller either resumes VM0 or waits until at least one of the spawned VMs has finished executing.

When VM0 reaches another function for which a faulty version exists, this process is repeated. It is not repeated if VM0 reaches the same function again or if one of the spawned VMs reaches a function with faulty versions. This ensures that at most one faulty version is active in any VM and none are active in VM0. When VM0 reaches the end of the workload and the controller detects that the experiment is complete, it waits for all spawned VMs to complete and terminates.

4.4 Evaluation

We evaluate our approach by applying it to seven real world Linux file systems. We provide a list of the file systems along with a description of our experiment setup and workloads in Section 4.4.1. The research questions we intend to answer in this evaluation are detailed in Section 4.4.2. Experimental results are reported in Section 4.4.3

4.4.1 Experiment Setup

We first describe the execution environment we use for our experiments. We then cover our evaluation targets and the workloads we use for our experiments.

Execution Environment

We conduct our experiments on the following two machines:

S_1: The first system is equipped with an AMD Threadripper 2990WX CPU with 32 physical and 64 logical cores, 128 GiB of RAM and a 1 TB NVMe SSD.

S_2: The second system is equipped with an AMD Threadripper 2970WX CPU with 24 physical and 48 logical cores, 64 GiB of RAM and a 1 TB NVMe SSD.

Both systems run Ubuntu 19.10. We use QEMU 4.0.0 as the VMM for our experiments, with KVM acceleration enabled. All SFI experiments are conducted on S_1, while S_2 is used to generate and build faulty versions and perform error propagation analysis.

Evaluation Targets

We apply our technique to 7 Linux file systems. An overview of the target file systems is given in Table 4.1. All file systems are extracted from Linux 5.0.

The virtual machines we use to run our SFI experiments are configured with 1 vCPU, 2 GiB of RAM and a `qcow2` disk that is used by QEMU for snapshots but not used by or visible to the guest system. All files required for our experiments with integrated kernel modules are placed in the `initramfs`, along with BusyBox 1.28.1. The VMs run Linux 5.0. We use a custom kernel configuration that supports the required VirtIO functionality used in our logging and controller implementation. To keep scheduling and timing non-determinism, and noise in general, on the system to a minimum, we disable preemption and run a tickless kernel. Since our approach assumes that each VM only has a single CPU core, we also disable SMP support in the kernel.

When we perform experiments using individually built faulty versions of kernel modules, placing all required files in `initramfs` would not be feasible, and regenerating the `initramfs` for each experiment execution would entail excessive per-execution overheads. Instead, we provide a read-only `virtfs` share to the VMs which contains all required kernel modules. For experiments comparing our approach to conventional SFI test execution with separate faulty versions, we use the same kernel as in experiments with our approach and we take a VM snapshot after boot but prior to workload start to avoid having to reboot the VM for each experiment.

Evaluation Workloads

Even though all our evaluation targets are file systems, we cannot use identical workloads across all of them. This is due to the fact that some of the included file systems are read-only, whereas others differ in their supported feature set. We use the following workloads in our experiments:

- *Read-write*: This is the workload we use for full-featured file systems. It encompasses module insertion, file system mounting, a variety of common file system operations such as directory listing, file creation and deletion, reading

Table 4.1: Overview of the Linux file system kernel modules used in the evaluation.

Module	Description	LOC
`hfsplus`	General purpose journaling read/write FS	9111
`isofs`	CD-ROM read-only FS	2922
`ntfs`	General purpose journaling FS, limited read/write	17021
`overlayfs`	Union mount read/write FS	7086
`romfs`	RomFS EEPROM read-only FS	722
`squashfs`	Compressed read-only FS	2791
`vfat`	General purpose read/write FS	6328

and writing, unmounting, and module removal. It is used for `hfsplus` and `vfat`.

- *Limited read-write*: We use this workload for our experiments with the `ntfs` file system as the kernel module does not support file creation. The workload resembles the regular read-write workload apart from the omission of the file creation step.
- *Read-only*: This is the workload we use for read-only file systems. It encompasses module insertion, mounting, common file system operations for read only file systems (i.e., directory listing and reading), unmounting, and module removal. This workload is used for `isofs`, `romfs`, and `squashfs`.
- *Overlay*: This is the specialized workload we use for the `overlayfs` file system. All other workloads operate on prepared file system images, but this is not applicable for the `overlayfs` file system. Therefore, we use this specialized workload. The functionality it exercises resembles the read-write workload, but it does not make use of a prepared image.

4.4.2 Research Questions

To evaluate the impact our proposed approach has on the performance and precision of kernel SFI and error propagation analysis, we investigate the following research questions:

RQ 1 Can our approach speed up sequential SFI test execution?

RQ 2 Can our approach speed up parallel SFI test execution?

RQ 3 How does our approach affect the number of executed faulty versions?

Table 4.2: Number of executed and activated faulty versions in each execution mode.

Module	Classic Execution				Integrated Execution			
	Executed		Activated		Executed		Activated	
	Abs	Rel %	Abs	Rel %	Abs	Rel %	Abs	Rel %
`hfsplus`	2885	100	1228	42.56	1814	62.88	1235	68.08
`isofs`	1519	100	799	52.6	1246	82.03	799	64.13
`ntfs`	7158	100	2432	33.98	4997	69.81	2438	48.79
`overlayfs`	4180	100	1527	36.53	2117	50.65	1527	72.13
`romfs`	289	100	250	86.51	272	94.12	250	91.91
`squashfs`	1107	100	654	59.01	929	83.92	654	70.4
`vfat`	3210	100	1468	45.7	1960	61.06	1468	74.9

RQ 4 Can our approach reduce build times for SFI experiments?

RQ 5 Does our approach affect SFI result validity?

RQ 6 How does our approach affect detection rates and false positives in error propagation analysis?

4.4.3 Results

In the following, we report our experimental results. Apart from the reported build times, all reported numbers are averages over three repeated executions.

RQ 1: Sequential Speedup

To determine whether our approach can speed up sequential SFI test execution, we compare sequential execution times between our approach and the conventional execution model using faulty versions with TrEKer instrumentation. The speedups achieved by our approach over the conventional execution model are shown in Figure 4.5. For sequential execution, the relevant numbers are the speedups reported above the bars for a degree of parallelism of 1 for each target file system. The speedups we achieve range from 1.32× for `squashfs` to 2.45× for `overlayfs`. As we do not make use of parallelism here, these speedups are entirely the result of the ability of our approach to execute fewer faulty versions and its ability to avoid the common prefix problem discussed in Section 4.3.3.

Our approach outperforms the conventional execution model for all file systems in our evaluation. We achieve speedups from 1.32× to 2.45× and conclude that our approach is capable of speeding up sequential SFI test execution.

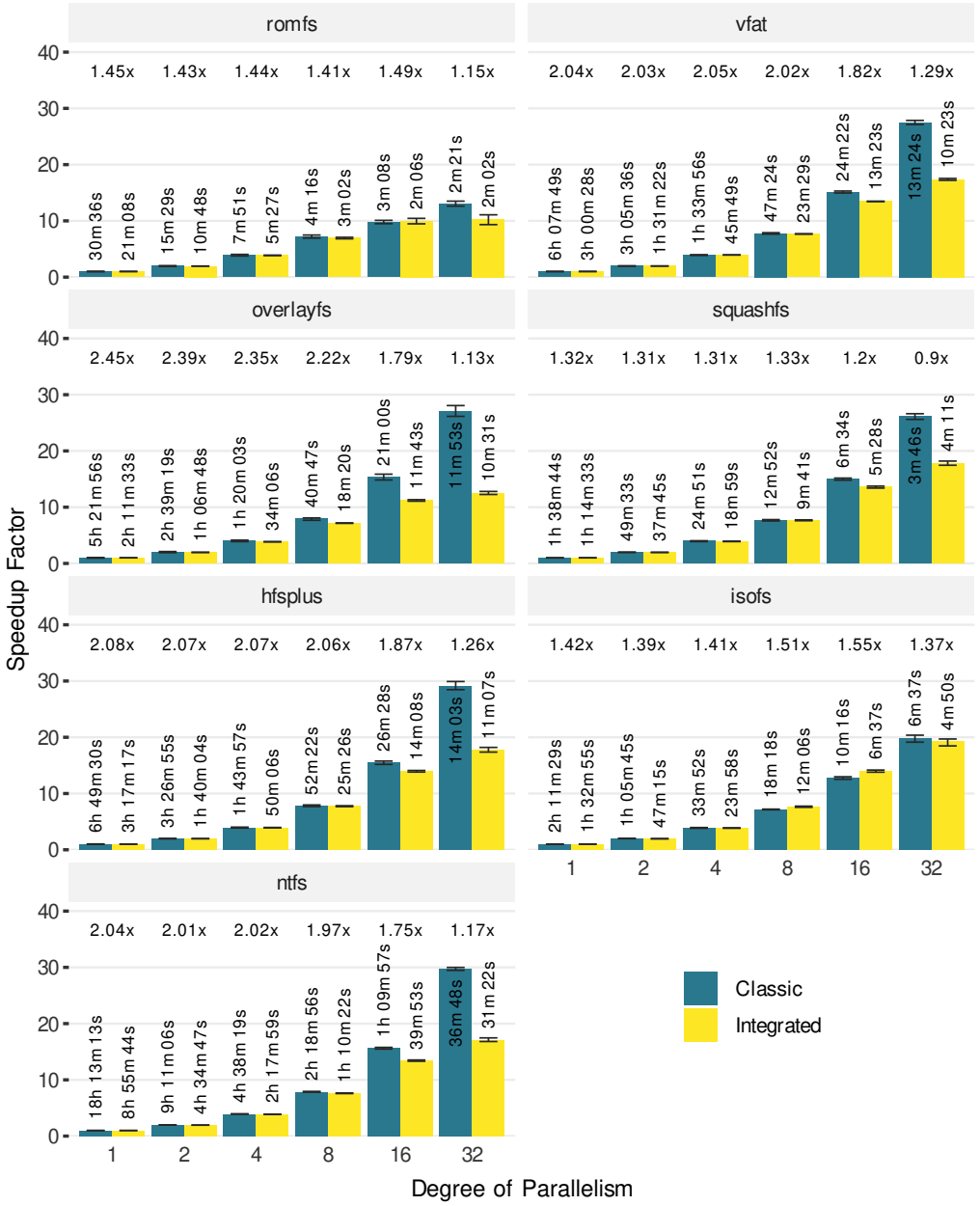


Figure 4.5: Execution times and relative speedups for SFI experiments based on wall time. Speedups compared to sequential execution are shown on the Y-axis. Absolute times are reported inside or above the bars. Speedups of integrated compared to conventional execution are shown at the top. All results are the average of three executions. The error bars indicate minimum and maximum values.

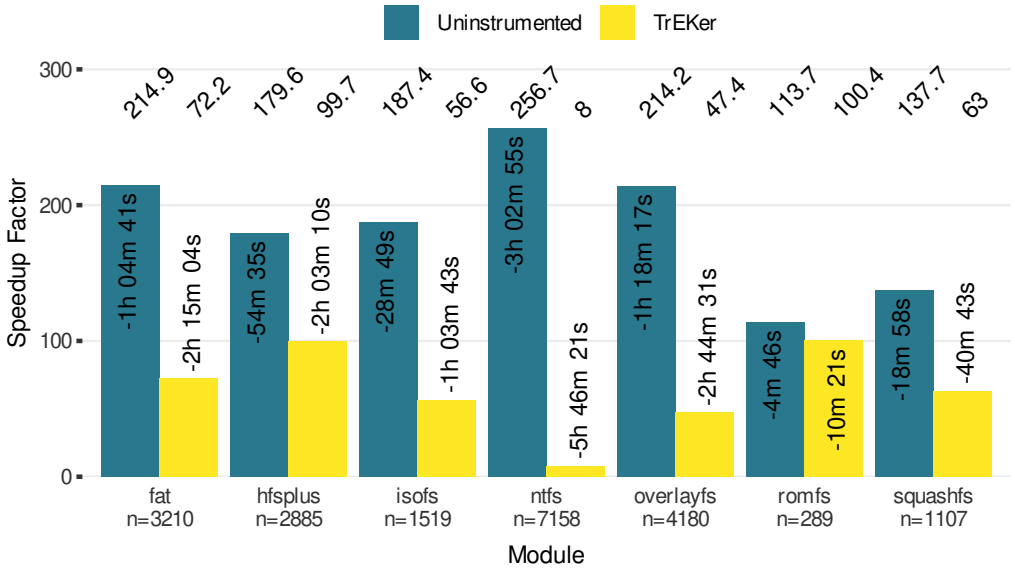


Figure 4.6: Build time speedups with and without instrumentation

RQ 2: Parallel Speedup

We investigate whether our approach is capable of accelerating parallel SFI test execution by comparing its performance to the conventional execution model across a range of different degrees of parallelism ranging from 1 to 32.

Figure 4.5 shows, for each file system and degree of parallelism, the speedup achieved relative to sequential execution in the same mode. The factors above the bars correspond to the speedups achieved by our approach over the conventional execution model at the same degree of parallelism. The times inside or above the bar give the absolute execution time.

We see that, except in one case, `squashfs` at a degree of parallelism of 32, our approach always outperforms the conventional execution model at the same degree of parallelism. However, it is also apparent that speedups relative to conventional execution at the same degree of parallelism reduce with increasingly parallel execution. This is due to the fact that, in some cases, our approach is not able to utilize the full computational resources available at higher degrees of parallelism throughout the entire SFI test execution. This can happen when, for example, the faulty versions for an SFI target are distributed across a large number of functions so that there are only a few faulty versions per function. In that case, our approach will not be able to fully utilize all available CPU cores with the faulty versions of a single function, and some cores will remain idle until the root VM (VM0) has reached the next function for which faulty versions exist. The conventional execution model, on the other hand, can simply run as many VMs in parallel as it has cores available. We note that, while wall time speedups for our approach are lower at higher degrees of

parallelism, user time speedups remain high since our approach reduces the overall amount of code to be executed, with `squashfs` at a degree of parallelism of 32 achieving a user time speedup of 15.2×.

Looking at speedups relative to sequential execution in the same mode as shown in Figure 4.5, it appears that our approach scales less well than the conventional execution model. This is for the reasons related to CPU utilization described above. However, our approach does achieve increasing speedups with increasing degree of parallelism for all file systems, with further potential for optimization in our prototype implementation.

We conclude that our approach is capable of accelerating parallel SFI test execution, although the benefits in execution time diminish for very high degrees of parallelism.

RQ 3: Executed Versions

We investigate how our approach affects the number of executed faulty versions as well as the number of activated faults by comparing the number of executed and activated faulty versions in the conventional execution model and in our approach as reported in Table 4.2. The table lists executed faulty versions, both absolute and as a percentage of all faulty versions, and activated faulty versions, absolute and as a percentage of executed faulty versions in the same mode. The conventional execution model requires execution of every faulty version. Therefore 100 % of faulty versions are executed for all file systems in this mode. This includes faulty versions with faults in functions that the workload does not trigger, resulting in activation rates below 60 % for all file systems except `romfs`, which, due to its straightforward code structure, achieves fairly high code coverage with our workload and a fault activation rate of 86.51 %. Our approach requires the execution of fewer faulty versions than the conventional execution model, ranging from 50.65 % to 94.12 %. This corresponds to a reduction in the number of executed faulty versions of 250 to 2438. As this reduction is the result of not executing inactive faulty versions, our approach also reaches higher activation rates ranging from 48.79 % to 91.91 %. While, in the conventional execution model, we only saw an activation rate above 60 % for a single file system, with our approach, all file systems bar one achieve such an activation rate.

We conclude that our approach can effectively reduce the number of faulty versions that need to be executed in SFI testing. We achieved fewer executed faulty versions and higher activation rates for all evaluated file systems.

RQ 4: Build Times

With this research question, we investigate the impact our approach has on build times for SFI experiments, both with the instrumentation required for kernel EPA

using our augmented version of TrEKer and without it. User build time speedups for all 7 file systems used in our evaluation, both instrumented and uninstrumented, are shown in Figure 4.6. Absolute time savings are listed vertically inside or above the bars. Speedup factors are shown on the Y-axis and above the bars. For all modules and in both modes, building an integrated module containing all faulty versions is substantially faster than building each faulty version separately. Note that we use incremental compilation in the latter case to minimize the work required for each faulty version. Speedups range from 8× to over 250×, with the instrumented case achieving lower speedups across all modules. This is particularly noticeable for `ntfs`, which is also the file system with the most faulty versions in our evaluation. We attribute this to inefficiencies in the instrumentation code and note that, although it achieves the lowest speedup, the instrumented `ntfs` build is also the one with the largest absolute time reduction.

With speedups ranging from 8× to 256×, we conclude that our approach can substantially reduce build times for SFI experiments.

RQ 5: Result Validity

We investigate the impact of our approach on SFI results using conventional failure mode detectors to check whether result validity is adversely affected. To that end, we report the observed results for integrated executions at different degrees of parallelism with activated faults and compare them to the results for conventional executions at the same degree of parallelism. Figure 4.7 shows the observed result distributions for all evaluation targets. We distinguish four common classes of SFI test results:

- *No Failure*: The test run finished without any error indication.
- *Timeout*: The test run did not finish within its execution time budget and was terminated by the experiment control logic.
- *Workload Failure*: The test run terminated with an error indication from user-mode, for instance, by aborting with an exit code indicating a failure.
- *Kernel Failure*: The test run terminated with an error indication from the kernel, such as a kernel panic.

We set the execution time budget of the timeout detector to 30 s for all runs since this value is considerably higher than the fault-free, but instrumented, execution time for all our target modules and should hence avoid premature timeout detections.

Across all modules and execution modes, “No Failure” is the most common result, with the exception of `romfs` for which “Workload Failure” is the most common. “Timeout” is the least common result with less than 10 % of runs having this

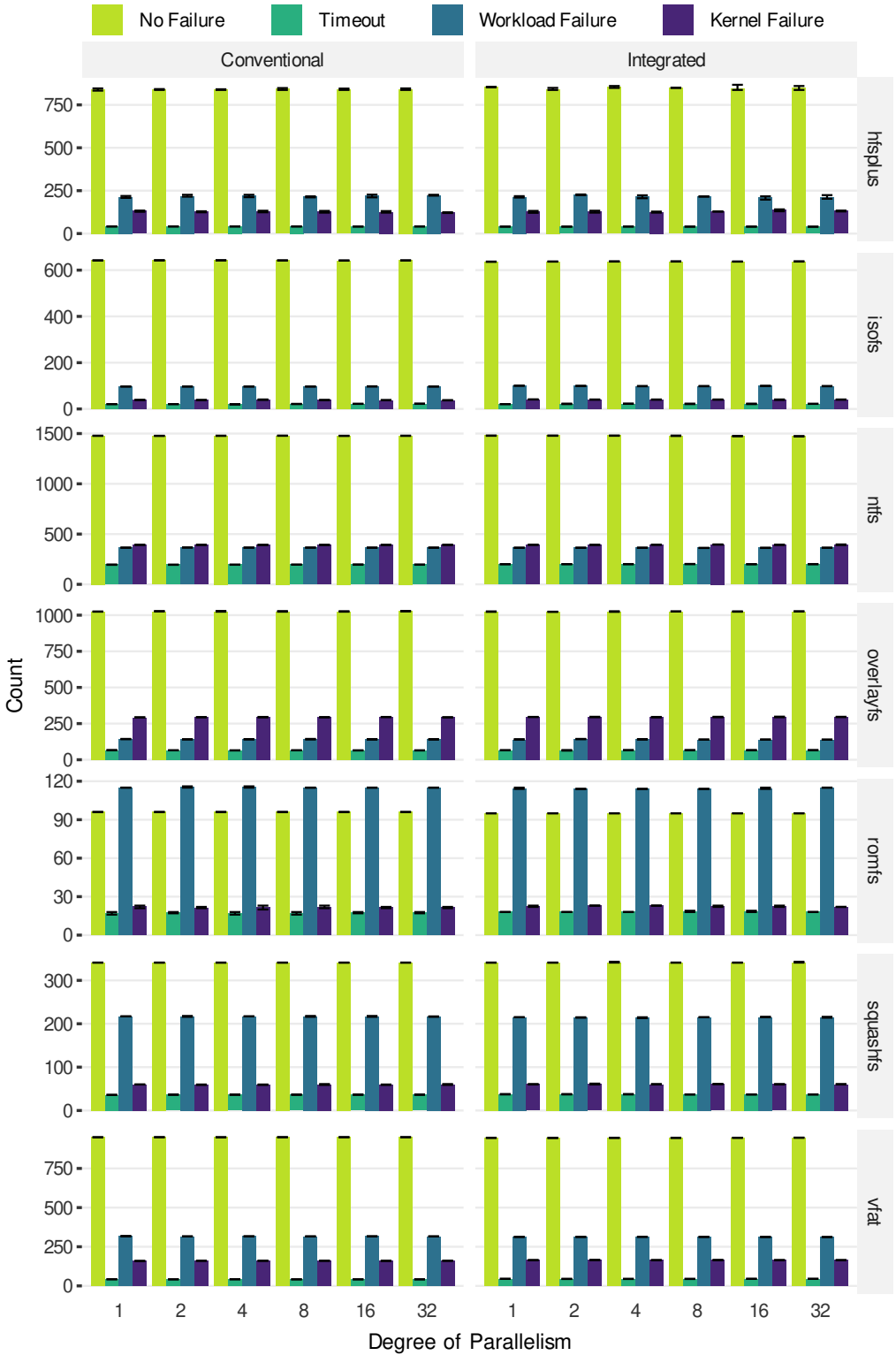


Figure 4.7: Result distributions for FI experiments.

Table 4.3: Results of χ^2-tests. p values and Cramer's V are reported for each test. The *Rej* column indicates if H_0 (no association between SFI results and execution mode) can be rejected (✓) or not (✗).

(a) χ^2-test results for degrees of parallelism from 1 to 4

	par = 1			par = 2			par = 4		
Module	p	V	Rej	p	V	Rej	p	V	Rej
`hfsplus`	0.943	0.0072	✗	0.786	0.0120	✗	0.818	0.0112	✗
`isofs`	0.943	0.0090	✗	0.957	0.0081	✗	0.963	0.0077	✗
`ntfs`	0.981	0.0035	✗	0.978	0.0037	✗	0.980	0.0036	✗
`overlayfs`	0.986	0.0040	✗	0.999	0.0018	✗	0.998	0.0021	✗
`romfs`	0.980	0.0110	✗	0.976	0.0119	✗	0.976	0.0119	✗
`squashfs`	0.990	0.0053	✗	0.987	0.0059	✗	0.983	0.0065	✗
`vfat`	0.900	0.0081	✗	0.888	0.0085	✗	0.916	0.0076	✗

(b) χ^2-test results for degrees of parallelism from 8 to 32

	par = 8			par = 16			par = 32		
Module	p	V	Rej	p	V	Rej	p	V	Rej
`hfsplus`	0.974	0.0055	✗	0.925	0.0080	✗	0.983	0.0048	✗
`isofs`	0.967	0.0074	✗	0.955	0.0083	✗	0.967	0.0074	✗
`ntfs`	0.953	0.0048	✗	0.974	0.0039	✗	0.969	0.0042	✗
`overlayfs`	0.981	0.0044	✗	0.979	0.0045	✗	0.969	0.0052	✗
`romfs`	0.965	0.0135	✗	0.967	0.0132	✗	0.981	0.0109	✗
`squashfs`	0.993	0.0049	✗	0.995	0.0043	✗	0.995	0.0043	✗
`vfat`	0.897	0.0082	✗	0.898	0.0082	✗	0.884	0.0086	✗

outcome, which suggests that out execution time budget choice was sensible. The result distributions remain stable across repeated runs and across different degrees of parallelism. We observe the largest variability for `hfsplus` ($P = 16$) in both execution modes with a maximum difference of result class counts of 30 (less than 3 %) between repeated executions. To assess whether the result distributions are significantly affected by integrated parallel or sequential execution when compared to conventional sequential execution, we conduct pairwise Pearson's χ^2-tests of independence between the conventional sequential execution and each integrated execution. We test the null hypothesis H_0 that "the obtained result distribution is independent from the execution mode", with the alternative hypothesis H_1 that "there is an association between result distribution and execution mode". We try to reject H_0 with a significance level of $\alpha = 0.05$. Table 4.3 reports the resulting p and Cramer's V values along with the decision whether we can reject H_0. Cramer's V is a measure of association based on the χ^2-statistic and ranges from 0 to 1, where the larger the

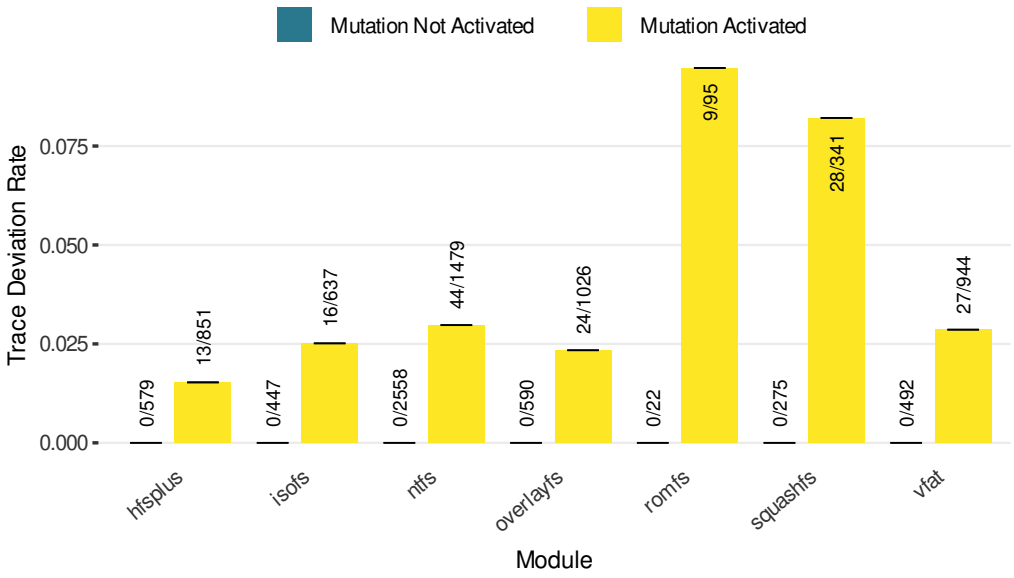
value, the stronger the association. With $p \gg \alpha$ for all modules across all execution modes, we fail to reject H_0. Hence, we cannot establish that there is an association between result distributions and execution mode. Accordingly, Cramer's V does not hint at association with $V < 0.015$ for all tests.

We conclude that integrated parallel or sequential execution does not affect SFI results of conventional oracles when compared to conventional sequential execution.

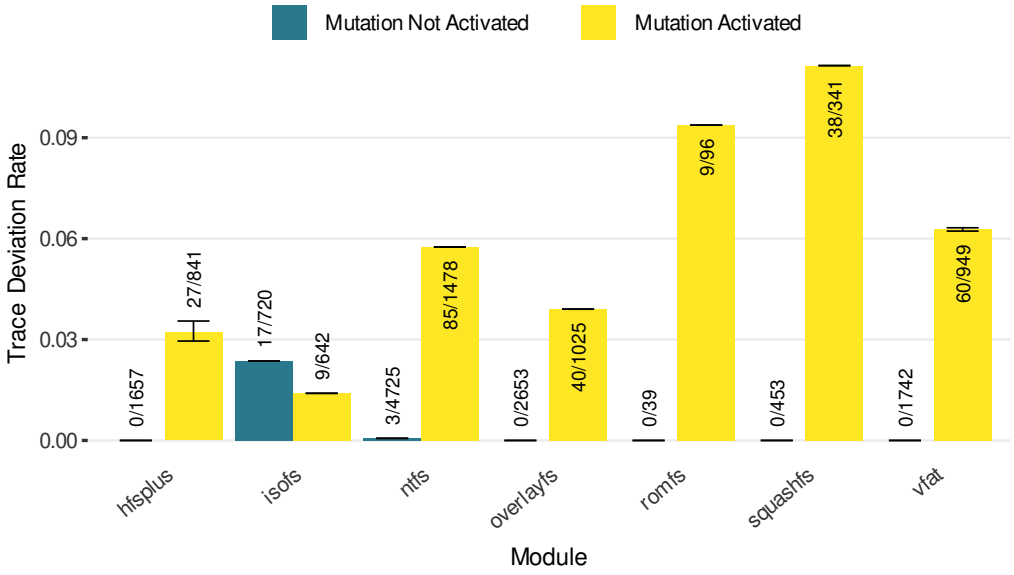
RQ 6: Detection Rates

We investigate how integrated execution affects the trace deviation rates detected by the TrEKer EPA. For this purpose, we analyze the execution traces that we collected during the sequential runs of our SFI experiments in both classic and integrated execution. As apparent from Figure 4.7, the "No Failure" class is the most common SFI outcome. More than 60 % of the runs with activated fault for all modules fall into this class, with the exceptions of `romfs` with 38 % and `squashfs` with 52 %. For error propagation analysis, this is the most interesting class of results as conventional oracles cannot detect any deviation despite the fault being activated. Either the injected fault is benign and has no effect or the effects have not manifested yet in a way that can be detected by conventional, external detectors. Both cases can be distinguished by EPA techniques such as TrEKer since execution trace deviations can be detected in the latter case. We report the TrEKer trace deviation rates for that case in Figure 4.8 with the bars labeled "Mutation Activated". In addition to the rates, the plots also contain the absolute numbers inside or above the bars. In order to assess the false positive detection rates, we also perform an analysis of "No Failure" run traces without activated fault, which are labeled accordingly in the plots. Any deviation detected in such a run cannot be caused by an injected fault and is therefore a false positive. The TrEKer analysis detects deviations by comparing execution traces against fault-free golden run traces, i.e., traces without injected faults. Since we took special care to create a low-noise and deterministic execution environment for our experiments, we use only a single golden run as comparison basis. In the case of the integrated experiments, this is the trace obtained from the root VM, so no separate golden runs need to be performed.

Overall, we observe low false positive rates, which are at 0 % across all modules when using integrated execution. In classic execution, we observe some false positives for `isofs` (2.4 %) and `ntfs` (below 0.1 %). The deviation rates with activated faults range from 1.4 % to 11.1 % for classic and from 1.5 % to 9.5 % for integrated execution. Overall, fewer deviations are detected with integrated execution for all modules, except for `isofs`. We attribute this reduction to less noise and less potential for spurious deviations in the execution traces due to the fine-grained VM snapshotting employed for integrated execution.



(a) Detected divergence rates with our approach.



(b) Detected divergence rates with the conventional approach.

Figure 4.8: Detected divergence rates for runs with and without activated faults.

With a 0 % false positive rate across all our target modules when comparing against a single golden run, we conclude that integrated execution is effective and does not increase false positive rates.

4.5 Discussion

As the investigation of our research questions in Section 4.4 shows, our approach is applicable to real world kernel code and can effectively accelerate sequential and parallel SFI testing without adversely affecting result validity. We therefore conclude that our approach enables effective SFI testing of kernel code on modern, parallel hardware.

We identify the following three main threats to the validity of our study:

1. The choice of target modules, system setup, configuration, and workload.
2. Limitations of the EPA approach used in our study.
3. Interactions between non-deterministic or timing-dependent behavior in the SUT and our snapshot-based experiments.

We evaluate our approach on seven widely used file systems from the Linux kernel using different workloads to exercise common file system functionality. However, kernel modules implementing different, unrelated functionality may behave differently. A different choice of workload may also yield different results. We tailor the configuration of the kernel on the target system to minimize noise and facilitate our VM cloning approach. To this end, we disable SMP support and preemption and run a small, BusyBox-based userspace. Different kernel configurations may increase scheduling noise, thereby affecting results or result stability. A more full-featured userspace, for example from a common desktop-focused Linux distribution, could also increase noise on the system and affect the experimental results. Our system configuration is a likely reason for the lower false positive rate we observed relative to the TrEKer experiments described in Chapter 2 using an augmented version of the same EPA approach. Such a target system may, for instance, require substantially more golden runs to achieve stable results. Our results may not generalize to other target modules, workloads, or configurations.

We use an augmented version of the TrEKer error propagation analysis in our experiments. As noted in Chapter 2, this approach restricts instrumentation and trace analysis in some respects to reduce overhead and improve performance. It may therefore miss behavioral divergences in some instances. The detection rates reported in Section 4.4.3 are subject to these restrictions. As we do not depend on the accuracy of the reported detection rates in our investigation of our other research questions, these limitations do not otherwise threaten the validity of our results.

Even though we have taken care to minimize noise and non-deterministic behavior on the system with our configuration, the SUT may still exhibit non-deterministic behavior between different executions, or timing-dependent behavior that leads to seemingly non-deterministic variations between executions. Our snapshot-based execution model may limit the first but could potentially increase the second. This can, in turn, influence the detection rates of the error propagation analysis. Since our approach does not result in any false positives for the modules used in our evaluation, this does not seem to occur in our evaluation, but, as noted above, other configurations may yield different results.

4.6 Conclusion

In this chapter, we introduced a novel approach for accelerating the SFI testing and error propagation analysis of operating system kernel code. Our approach speeds up SFI test execution in three ways: We avoid redundant code execution, automatically skip the execution of inactive faulty versions, and facilitate parallelization of SFI experiment execution. Moreover, our approach substantially reduces build times by integrating all faulty versions into a single module, thereby avoiding redundant compilation effort.

In our evaluation on seven widely used Linux file system implementations, we achieve sequential speedups of up to 2.45×, parallel speedups of up to 36.8× using 16 parallel instances, and build time speedups of up to 100.4× with instrumentation and 256.7× without. We find that our approach does not adversely affect SFI result validity and does not increase the false positive rate in error propagation analysis.

5 Fuzz Testing

Fuzzing is a form of random testing that is widely used for finding security and dependability issues. Approaches that leverage information about the control flow of prior executions of the program under test to decide which inputs to mutate further have proven particularly effective in practice. However, by relying solely on control flow information to characterize executions, such approaches may miss relevant differences. We saw in our work on TrEKer, which we describe in Chapter 2, how memory instrumentation can be useful in characterizing the runtime behavior of software components, and on this basis we propose augmenting evolutionary fuzzing by additionally leveraging information about memory accesses performed by the target program. The resulting approach can leverage more sophisticated information about the execution of the target program, enhancing the effectiveness of the evolutionary fuzzing. We implement our approach as a modification of the widely used AFL fuzzer [Zal] and evaluate our implementation on three widely used target applications. We find distinct crashes from those detected by AFL for all three targets in our evaluation. The contents of this chapter are, in parts verbatim, based on material from [CSS19].

5.1 Introduction

Fuzzing is an established form of random testing that has proven to be highly capable of finding bugs and vulnerabilities in numerous widely used programs and applications. It is commonly used to examine application software, libraries, as well as system level code and has been applied in practice to a wide variety of targets, ranging from smart contracts [JLC18] to operating system kernels [Sch+17; Vyu]. In recent years, fuzzing has emerged as an effective technique for finding bugs that involve memory safety violations for software written in languages such as C or C++ that do not guarantee memory safety. This category of software bugs is of particular interest as such bugs frequently have not just robustness but also security implications. Some of the most well known vulnerabilities of recent years belong to this category [MITa; Orm; Syn].

Fuzzing is a fairly broad category that covers a variety of approaches, including techniques that generate inputs from scratch, for instance based on a grammar specification, as well as techniques based on input mutation. It also encompasses both white and black box approaches. A widely used class of fuzzers are *coverage-guided,*

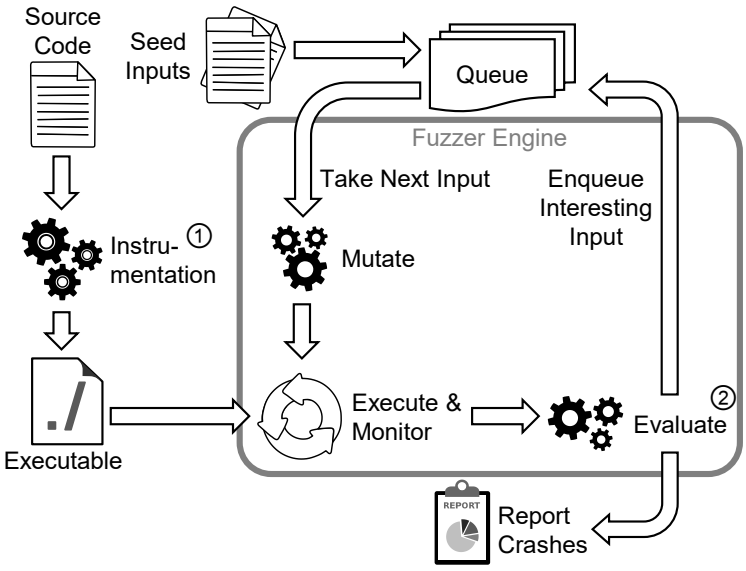


Figure 5.1: Evolutionary Fuzzing

evolutionary fuzzers. A typical fuzzing workflow from this category is illustrated in Figure 5.1. It consists of the following steps:

a) A set of seed inputs is selected (for instance, from the test suite of the program under test) and used to initialize the input queue.

b) The program under test is instrumented, typically during compilation, to gather coverage information at runtime. This step is marked ① in Figure 5.1.

c) The fuzzer picks an input from the queue, mutates it, and runs the instrumented program under test with the mutated input while monitoring its execution.

d) The resulting coverage information is evaluated to determine whether the program under test exhibited *interesting* behavior with the latest input. Typically, this means checking whether the program crashed or new edges in the control flow graph or new basic blocks were reached during that execution. This step is marked ② in Figure 5.1. Non-crashing, interesting inputs are put back into the queue for further mutation, whereas crashing inputs are reported.

The last two steps of this process (gray box in Figure 5.1) are repeated as often as possible with the available time budget, which is set by the user. Fuzzing typically emphasizes speed of individual executions and favors efficiency over complex program analysis techniques. Therefore, the inserted instrumentation (①) should be lightweight and incur little runtime overhead; and the evaluation (②) should be

```
1  int main(int argc, char **argv) {
2     int buffer[512] = {0};
3     FILE* fp = fopen(argv[1], "r");
4     int a = 0, b = 0, c = 0, d = 0;
5     fscanf(fp, "(%d, %d); (%d, %d)",
6           &a, &b, &c, &d);
7     int x = (a * b) & 511;
8     int y = (c * d) & 511;
9     printf("%d %d %d\n", buffer[x],
10          buffer[y], buffer[x + y]);
11    return 0;
12 }
```

Figure 5.2: A Motivating Example for MemFuzz

fast to allow as many executions of the target program as possible in the given time budget.

For these reasons, widely used coverage-guided, evolutionary fuzzing techniques generally rely on basic block or edge coverage data: It can be gathered with relatively little runtime overhead and compact representations enable quick comparisons in the evaluation step.

Relying on such control-flow centered coverage information to distinguish between executions of a fuzzing target is an efficient approach with a proven track record. However, relying solely on edge or basic block coverage information may lead to missing important differences in how programs process different inputs. For instance, consider the example shown in Figure 5.2. The presented program reads four integers from an input file using `fscanf` (lines 3–6). Crucially, it does not perform any error checking and silently falls back to default values in the presence of invalid input. It then uses these integers to compute indices (lines 7 and 8) which are used to access a fixed-size buffer (lines 9 and 10). The last of these accesses (`buffer[x + y]` in line 10) is potentially out of bounds, which constitutes a memory safety violation bug.

This bug is straightforward to find by manual inspection and easy to detect using more heavyweight techniques like symbolic execution. Conventional coverage-guided, evolutionary fuzzing, however, struggles to find a crashing input for this program: There is only one path through the program, so the edge coverage such techniques rely on fails to provide meaningful feedback. Without any feedback, the resulting blind fuzzing is unlikely to produce a crashing input entirely by chance.

We fuzzed the program shown in Figure 5.2 using AFL 2.45b [Zal], a widely used, state of the art fuzzing tool, and a seed input `(1, 1); (1, 1)` for more than 24 hours without finding a crash. This is due to the fuzzer trimming the seed input prior to mutating it and, being entirely reliant on edge coverage, discarding most

of it. Absent a working feedback mechanism, the fuzzer would then need to reconstruct the structure of the input entirely by chance to find a crash, which is highly unlikely.

Similar issues can arise due to the use of conditional moves or complex address calculations. Generally, the effectiveness of conventional, coverage-guided fuzzing is limited for code in which safe executions and unsafe, crashing executions exist with the same control flow, and in which the mutation of multiple parts of the input is necessary to generate an unsafe input from a safe one. In such cases, any intermediate inputs would be discarded by the fuzzer without further mutation as no new control flow behavior is exhibited, and the fuzzer would fail to find a crashing input.

We propose addressing this blind spot by using information about the memory addresses a program accesses, either in addition to or instead of edge coverage, to characterize executions. The resulting approach, which we call MemFuzz, should be able to distinguish between executions with the same control flow, based on differences in the accessed memory addresses. We implement a prototype of this approach based on AFL 2.45b. To summarize, we make the following contributions:

- A new characterization of program executions for feedback-guided fuzzing
- MemFuzz, an instrumentation and static analysis approach to allow for efficient collection of memory coverage data
- A prototype implementation based on the state of the art AFL fuzzer
- An extensive evaluation of the proposed approach, highlighting the strengths and limitations of the MemFuzz approach

The rest of this chapter is structured as follows: We discuss related work in Section 5.2. Our approach and implementation are presented in Section 5.3 and evaluated in Section 5.4. We discuss our results and threats to validity in Section 5.5 and present concluding remarks in Section 5.6.

5.2 Related Work

Fuzzing is a wide, active research area, which renders a comprehensive overview infeasible. Therefore, we focus on a discussion of how our work relates to other research on evolutionary fuzzing and omit an in-depth discussion of other related techniques (e.g. approaches using symbolic execution).

Recent research in this area aims at improving the various steps of the evolutionary fuzzing process. We focus our discussion on related work dealing with seed selection and minimization, instrumentation, and guidance mechanisms in this section. Recent work on search strategies, scheduling, and queue prioritization [BPR18;

CC18; Lem+18; LS18; Raw+17] is largely orthogonal to our approach as it does not alter how executions are distinguished. Therefore, we omit a detailed discussion.

5.2.1 Seed Selection

The selection of seed inputs is usually one of the first steps in fuzzing workflows (cf. Figure 5.1) and one that requires substantial manual effort. Research in this area has dealt with automating this step, achieving effective seed corpora by optimizing seed selection [Reb+14] or generation [Wan+17]. As our proposed approach does not involve alterations to the seed selection step, most of this work is orthogonal to ours.

However, it is common to minimize seed test cases prior to fuzzing as the mutation and execution of smaller inputs is more efficient. AFL [Zal], being the most widely used fuzzer in this area, even trims new inputs on its own to reduce their size and additionally ships with a separate tool, `afl-tmin`, to perform more sophisticated test case minimization.

Other approaches to test case minimization can also be applied to minimize seed inputs for fuzzing. Groce et al. propose an extension of delta debugging [Zel02] to support test case minimization while retaining arbitrary properties of executions of a given target program [Gro+16]. When used for seed minimization in the context of guided fuzzing, such approaches, which minimize test cases based on coverage information, may benefit from additionally applying memory instrumentation to avoid cases like the one discussed in the example in Figure 5.2 in Section 5.1. The MemFuzz modifications we propose for AFL involve changing the way the fuzzer trims new inputs to avoid this issue (but not the standalone `afl-tmin` tool). Other test case minimization tools could be adjusted in a similar way to leverage the MemFuzz memory instrumentation.

5.2.2 Instrumentation and Guidance

Research on instrumentation and guidance mechanisms for fuzzing either deals with devising new mechanisms or optimizing existing approaches for gathering coverage information or with the collection and usage of entirely different types of information to guide the fuzzing process.

Petsios et al. propose SlowFuzz [Pet+17] to automatically detect algorithmic complexity vulnerabilities. Like MemFuzz, SlowFuzz employs a different guidance mechanism from conventional evolutionary fuzzing, which does not exclusively rely on a notion of edge coverage. Instead, SlowFuzz uses resource usage, specifically the number of executed instructions, to guide the fuzzing process.

Hsu et al. propose InsTrim [Hsu+18], an approach for reducing the number of basic blocks that need to be instrumented to gather edge coverage information in order to reduce runtime overhead. We pursue a similar objective with MemFuzz's

static analysis described in Section 5.3.2, but for memory access instrumentation rather than basic block instrumentation.

Gan et al. tackle the problem of *edge collisions* in AFL. The original AFL fuzzer may assign the same identifier to multiple edges, which results in inaccurate coverage information. Their solution, CollAFL [Gan+18], outperforms AFL in terms of coverage and crashes found.

AFLGo [Böh+17] is a modification of AFL for directed fuzzing that aims to extensively fuzz specific parts of the program under test for use cases like patch testing. AFLGo extends AFL's feedback and guiding mechanism to consider the distance to the targeted parts of the program under test.

Other approaches do not investigate different types of information as basis to guide the fuzzing but different ways of gathering edge coverage information. For instance, PTFuzz [Zha+18] and kAFL [Sch+17] rely on the Intel Processor Trace functionality of newer Intel x86 CPUs to obviate compile-time instrumentation and dynamic binary instrumentation. Such optimizations are not applicable to the instrumentation MemFuzz requires as the mechanisms they use are restricted to control flow tracing.

5.3 Approach

In this section, we present the design and implementation of MemFuzz, our approach to distinguish meaningfully different program executions with the same control flow based on the sets of memory addresses used in *read* and *write* memory accesses. MemFuzz provides novel guiding strategies to enhance the conventional coverage-guided evolutionary fuzzing and is implemented as an extension to the widely used AFL fuzzer. It requires a compile-time instrumentation of the targeted programs, which we discuss in Section 5.3.2, as well as certain runtime support, which we discuss in Section 5.3.3. MemFuzz extends the information the fuzzing feedback loop can rely on to guide the fuzzing process, which we detail in Section 5.3.4. We begin our discussion with an overview of MemFuzz and its integration with AFL in Section 5.3.1.

5.3.1 Overview

The overall design goal of MemFuzz is closing the blind spot of traditional coverage-guided fuzzing, which we illustrated with the example in Figure 5.2, where executions cannot be meaningfully distinguished by means of control flow only. In order to be of practical use, the MemFuzz implementation must be simple to apply to new targets and maintain sufficient fuzzing performance. This is why we designed MemFuzz as extension to the AFL fuzzer, which is widely used and known to perform well.

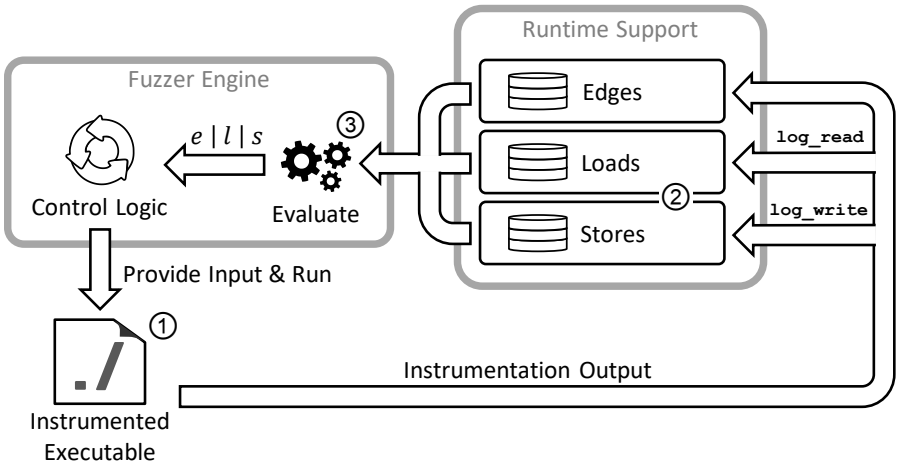


Figure 5.3: MemFuzz: Design Overview

Figure 5.3 provides an overview of the MemFuzz design, which consists of three essential parts:

- Compile-time instrumentation, marked ① in the figure;
- A runtime support library, marked ②;
- The fuzzer feedback mechanism that categorizes an execution as interesting or uninteresting, marked ③.

The MemFuzz instrumentation, which focuses on loads and stores, provides information on performed memory accesses in addition to AFL's edge coverage information. The accessed memory addresses are tracked within the runtime support library that employs two bloom filters to efficiently track which addresses have been accessed to read or write memory. This extended pool of coverage information is then used by the fuzzer engine to evaluate the past execution and decide whether the used input is eligible for further mutation. The MemFuzz extension to AFL's evaluation mechanism allows the usage of all three information sources either separately or in certain combinations.

5.3.2 Instrumentation

To capture the addresses used in *read* and *write* memory accesses, we instrument such accesses at compile-time. As AFL already includes an LLVM-based mode and ships with an LLVM-instrumentation pass, we chose to build onto that feature for MemFuzz. In the LLVM intermediate representation (LLVM-IR), memory accesses are explicit and can only be performed using a small number of specific instructions, such as `load` for reading and `store` for writing memory locations. It is therefore

especially well suited for the instrumentation of memory accesses as all instrumentation sites are apparent, a fact that we also took advantage of in our work on TREKER as described in Section 2.4. Moreover, instrumenting memory accesses at compile time incurs lower runtime overhead than techniques using, e.g., dynamic binary instrumentation, and allows us to perform static analysis as discussed in Section 5.3.2.

Basic Memory Instrumentation

We start from the existing AFL LLVM instrumentation pass and enhance it to also instrument `load` and `store` instructions. By extending the existing pass, we ensure backwards compatibility as we leave the original edge instrumentation functionality untouched. The enhanced instrumentation pass inserts calls to logging functions, which reside in the support runtime library (cf. Section 5.3.3), prior to loads and stores. An example of the memory instrumentation is shown in Figure 5.4. The small C function in Section 5.3.2 reads from a computed memory location (`d[i]`) and writes to the global variable `g`. Section 5.3.2 lists a simplified version of the corresponding LLVM-IR. Note that not only the read and write accesses are explicitly represented (lines 5 and 6) but also the address calculation (line 4) for the read access. The possible instrumentation sites are the `load` in line 5 and the `store` in line 6. The listing in Section 5.3.2 shows the LLVM-IR after the MEMFUZZ instrumentation has been applied. Here, the `load` is instrumented by the insertion of a call to `log_read` (line 6) prior to the actual `load`. Note that the additional `bitcast` (line 5) is needed to correctly call `log_read` as type casts in LLVM are explicit, but it does not impose actual runtime overhead. The `store` in line 6 is not instrumented in this example due to MEMFUZZ's static analysis optimization that we detail in Section 5.3.2. The `log_read` function takes the address of the memory location the program is reading as its argument. The value that is read is not logged. `store` instructions are treated analogously using the `log_write` function.

Instrumentation Site Filtering

At runtime, MEMFUZZ-instrumented code incurs the overhead of an additional function call as well as the actual logging implementation (cf. Section 5.3.3) for each instrumentation site. It is therefore desirable from a performance standpoint to avoid instrumentation where possible. To this end, our instrumentation pass includes a static analysis component that excludes instrumentation sites that correspond to certain classes of memory accesses that can be safely ignored without losing information beneficial for the guiding of the fuzzing process (i.e., memory accesses for which the address is not computed using the program input). For instance, a `load` from a global variable will use the same address for every access. Instrumenting such an input-independent memory access does not yield any information beyond the fact that the access has taken place. This fact, however, is already more effi-

```
1 int g = 0;
2
3 void f(int* d, long i) {
4   g = d[i];
5 }
```

(a) C-Code

```
1 @g = global i32 0
2
3 define void @f(i32*, i64) {
4  %4 = gep i32, i32* %0, i64 %1
5  %5 = load i32, i32* %4
6  store i32 %5, i32* @g
7  ret void
8 }
```

(b) LLVM-IR Representation

```
1  @g = global i32 0
2
3  define void @f(i32*, i64) {
4   %4 = gep i32, i32* %0, i64 %1
5   %5 = bitcast i32* %4 to i8*
6   call void @log\_read(i8* %5)
7   %6 = load i32, i32* %4
8   store i32 %6, i32* @g
9   ret void
10 }
```

(c) LLVM-IR with Memory Instrumentation

Figure 5.4: A simplified example showing the operation of the instrumentation pass and the static analysis component.

ciently detectable by means of control flow instrumentation. Our static analysis therefore excludes instrumentation sites that correspond to accesses to globals from instrumentation. The `store` in Section 5.3.2 (line 6) is an example of this, and consequently, there is no call to `log_write` prior to the `store` in Section 5.3.2. Additionally, we also exclude accesses to stack variables allocated in the current frame unless they use a dynamically computed offset.

A simplified version of the algorithm we use to decide whether to filter out a given instrumentation site is given in Figure 5.5. Note that the pseudocode omits loop detection and result caching. The analysis is intraprocedural and conservative, which allows us to substantially reduce the number of instrumentation sites without incurring large compile time overhead or erroneously skipping relevant instrumentation sites.

5.3.3 Runtime

The `log_read` and `log_write` functions that are invoked by the inserted instrumentation code as described earlier are implemented within a runtime support library. The two functions implement the actual tracking of memory addresses. As the instrumentation and the runtime library are decoupled, a wide variety of design choices for the implementation of these functions are viable. As we aim to maintain backwards compatibility with AFL's edge coverage instrumentation, we extend AFL's runtime library with the runtime support functions our MemFuzz instrumentation requires. Moreover, we choose to focus on performance and favor simplicity in our implementation. The runtime library therefore only keeps track of the sets of memory addresses that are used to read or write memory locations. It does not track the order of accesses or the number of accesses to the same address. In order to avoid the need for heap allocations in the runtime, we employ bloom filters to represent the sets of memory addresses. As these are fixed-size data structures, memory overhead is independent of the number of accesses. Furthermore, adjusting bloom filter parameters allows tuning of the runtime for different application scenarios or target applications. Since the runtime support library and the fuzzing target share the same address space, avoiding heap allocations in the runtime prevents such allocations from affecting the memory layout or addresses used by the target application. Since bloom filters do not allow the retrieval of set members, it is not possible to reconstruct the exact set of addresses that were accessed during an execution afterwards. However, as we will discuss in Section 5.3.4, this is not a drawback in this application context as we only need to decide whether a memory address has been seen before in an execution. We use the `xxHash64` [Col] algorithm for our bloom filter implementation.

```
1:  procedure SkipAddr(a: Addr)
2:      if IsGlobal(a) then
3:          return true;
4:      else if IsLocalAlloca(a) then
5:          return true;
6:      else if IsCast(a) then
7:          iv ← IncomingValue(a);
8:          return SkipAddr(iv);
9:      else if IsGEP(a) then
10:         return SkipGEP(a);
11:     else if IsPHI(a) then
12:         return SkipPHI(a);
13:     else
14:         return false;
15:     end if
16: end procedure

17: procedure SkipGEP(g: GEPInst)
18:     if HasSafeConstantOffset(g) then
19:         a ← IncomingValue(g);
20:         return SkipAddr(a);
21:     else
22:         return false;
23:     end if
24: end procedure

25: procedure SkipPHI(p: PHIInst)
26:     for all iv ∈IncomingValues(p) do
27:         if ¬SkipAddr(iv) then
28:             return false;
29:         end if
30:     end for
31:     return true;
32: end procedure
```

Figure 5.5: Instrumentation Site Filtering

5.3.4 Fuzzer

Our instrumentation and runtime provide the fuzzer with additional feedback for each execution. In addition to the edge coverage information, which the original AFL implementation relies on exclusively, our MemFuzz enabled fuzzer can use the two bloom filters containing `load` and `store` addresses to choose which inputs to mutate further. As discussed in Section 5.3.3, bloom filters do not allow the retrieval of entries. However, by keeping track of the bloom filters seen during previous executions of the target program, the fuzzer can tell whether an execution has resulted in a reading or writing memory access that has not been seen during any previous execution. This is necessarily the case if any bit in the bloom filter is set that has not been set during any prior execution.

After each execution of the target program, there are three predicates on the execution available to the fuzzer:

- Whether the input resulted in new edge coverage (e);
- whether any previously unseen memory addresses were written to (s); and
- whether any previously unseen memory addresses were read from (l).

We use this information to determine whether an execution exhibited novel behavior, and therefore, whether the corresponding input should be mutated further. We do not change other parts of the fuzzer such as queue prioritization.

MemFuzz allows the use of any boolean expression over the aforementioned three predicates to be used as a novelty criterion by the fuzzer. We call such expressions strategies. Our prototype implementation supports both a conjunction or disjunction over any subset of predicates. As our goal is a more fine-grained rather than coarse-grained distinction between executions, we focus on logical disjunctions (e.g. $e \vee s \vee l$ or $s \vee l$) as guiding strategies. With such strategies, the fuzzer considers an execution to have exhibited novel behavior if, for instance, it resulted in new edge coverage *or* a store to a new memory address. As a result, the set of executions considered novel by any such strategy that also takes edge coverage into account is a superset of the set of executions considered novel by conventional coverage-guided fuzzing. This is in line with our goal of a more fine-grained distinction.

Table 5.1: Overview of Evaluation Targets

Application	Version	Description
`ffmpeg`	2.0.1	Audio/video processing
`ImageMagick`	6.7.5-10	Bitmap image processing
`libxml2`	2.7.0	XML parser library/toolkit

Table 5.2: Seed Inputs and Dictionary Tokens for each Evaluation Target

Application	Dictionary Tokens	Seed Inputs	Seed Input Sizes
`ffmpeg`	992	196	55 B – 99 KiB
`ImageMagick`	90	15	41 B – 262 KiB
`libxml2`	60	69	5 B – 40 KiB

5.4 Evaluation

To evaluate the applicability and effectiveness of the proposed approach, we apply our prototype implementation to three evaluation targets and investigate the following research questions:

RQ 1 Does MemFuzz find different crashes from conventional, coverage-guided evolutionary fuzzing?

RQ 2 What runtime overhead does our prototype implementation impose?

RQ 3 How does MemFuzz affect the edge coverage of the generated inputs?

RQ 4 How much can the static analysis component reduce the number of necessary instrumentation sites?

In the following, we first describe our experimental setup and evaluation targets in Section 5.4.1. Then, we address the four research questions in Sections 5.4.2 to 5.4.5.

5.4.1 Experimental Setup

In the following, we describe our choice of evaluation targets, the execution environment in which we conduct our experiments, and the configurations we use in fuzzing our target applications.

Table 5.3: Overview of Fuzzing Configurations

Designation	Fuzzer	Strategy	ASAN	Seed/Dict
AFL	AFL	Edge	No	Null
AFL+A	AFL	Edge	Yes	Null
MEM	AFLm	Mem	No	Null
MEM+A	AFLm	Mem	Yes	Null
HYB	AFLm	Mem + Edge	No	Null
HYB+A	AFLm	Mem + Edge	Yes	Null
AFL^S	AFL	Edge	No	Seed & Dict
AFL+A^S	AFL	Edge	Yes	Seed & Dict
MEM^S	AFLm	Mem	No	Seed & Dict
MEM+A^S	AFLm	Mem	Yes	Seed & Dict
HYB^S	AFLm	Mem + Edge	No	Seed & Dict
HYB+A^S	AFLm	Mem + Edge	Yes	Seed & Dict

Evaluation Targets

We evaluate our approach on the three target applications listed in Table 5.1. The table gives a brief description of each target and lists the used versions. All three targets are widely used, parse complex file formats and are primarily implemented in C, which does not guarantee memory safety. We selected older versions of our targets to ensure that the fuzzers used in the evaluation are able to find crashes in a reasonable amount of time.

Execution Environment

We use machines with an Intel Core i7-4790 CPU, 16 GiB of RAM, and a 500 GB SSD running Debian 8.10 with a distribution-provided Linux 4.9 kernel for all experiments.

Experiment Configurations and Execution

Table 5.3 lists all configurations we use for fuzzing each of our targets. We use the original AFL fuzzer in version 2.45b (being the version our implementation is based on) that relies exclusively on edges (predicate e, cf. Section 5.3.4) as guiding strategy in the AFL configurations. For our modified AFL (AFLm in the table), we distinguish two configurations that use different guiding strategies: The first, MEM, relies exclusively on memory accesses to distinguish interesting executions ($l \vee s$). The second, HYB, is a hybrid strategy taking both edges and memory accesses into account ($l \vee s \vee e$). Additionally, we fuzz our targets both with and

without AddressSanitizer (ASAN) [Ser+12] as well as with simple Null seed inputs without dictionaries and with a larger corpus of seed inputs and dictionaries to cover a variety of realistic scenarios. For the configurations using seed inputs and dictionaries, the number and size of the seed inputs and the number of tokens for the fuzzing dictionary are listed in Table 5.2. Overall, this amounts to a total of 12 distinct fuzzing configurations.

We execute each of these 12 configurations for each target for three hours using eight parallel instances (one master and seven secondary instances, totaling 24 h computation time). This process is repeated five times for a total of 180 experiments.

Dictionaries and Seed Inputs

Our Null seed input is identical for all targets and consists of a single file containing one byte with value `0x00`. For the seeded configurations, we construct an individual corpus of seed inputs and a token dictionary for each target (cf. Table 5.2). The seed input corpus for `ImageMagick` and `libxml2` is constructed from test case and example files that ship with the respective target. For `ffmpeg`, the seed input corpus is constructed from small sample files from the *FFmpeg Automated Testing Environment (FATE)* [FFm]. Starting with these files, we applied AFL's corpus and file minimization tools `afl-cmin` and `afl-tmin` to remove redundant input files and file contents which do not contribute to interesting program behavior in terms of edge coverage.

The token dictionaries were constructed by merging the dictionaries for the file types relevant for the respective targets that are included with AFL. The only exception is `ffmpeg` since AFL does not include dictionaries for video and audio files. We used `libtokencap`, which ships with AFL, on the samples from FATE to automatically extract tokens for all file types included in the corpus.

5.4.2 RQ 1: Crashes

To determine whether the MemFuzz approach is capable of finding different crashes compared to conventional, coverage-guided evolutionary fuzzing, we analyze the crashing inputs generated by each fuzzing configuration for each target. We re-use the same target binaries that were used for `AFL+A`, as they are built with AddressSanitizer, to reproduce all crashes and gather stack traces. We then bucket crashes using an enhanced form of stack hashing. We hash call stack addresses, program counter, stack pointer as well as base pointer values, and the cause of the crash as reported by AddressSanitizer (read, write, or signal) using the `xxHash64` algorithm. In case of crashes caused by signals, we also include the signal type (e.g. `SEGV` or `ABRT`) but not the exact memory addresses triggering an AddressSanitizer error or signal. The additional information beyond the call stack and program counter is

included to increase precision. We exclude memory addresses to ensure that this technique does not give an advantage to our approach.

The described stack hashing yields a set of hashes for each combination of fuzzing configuration and target (36 sets in total) where each hash corresponds to a distinct crash. We compare these sets for each target and consider all crashes found in at least one of the five repetitions. The results are visualized in Figures 5.6 to 5.10 as UpSet plots [CLG17; Lex+14]. The UpSet plots show the number of distinct crashes (horizontal bars at the bottom left), the set intersections (connected circles at the bottom), and the intersection sizes (vertical bars at the top). Note that the intersections are exclusive and form a partition of the set of all crashes. We omit plots for cases in which there are no set intersections, e.g. because only one fuzzer found any crashes.

For `ffmpeg`, Section 5.4.2 shows that, without the use of AddressSanitizer, both MEM and HYB found one crash. In the same runtime, AFL found five different crashes, but not the one found by MEM and HYB. With AddressSanitizer (Section 5.4.2), on the other hand, neither HYB+A nor MEM+A were able to detect crashes not found by AFL+A. In both configurations, MEM and HYB find the same crashes as each other.

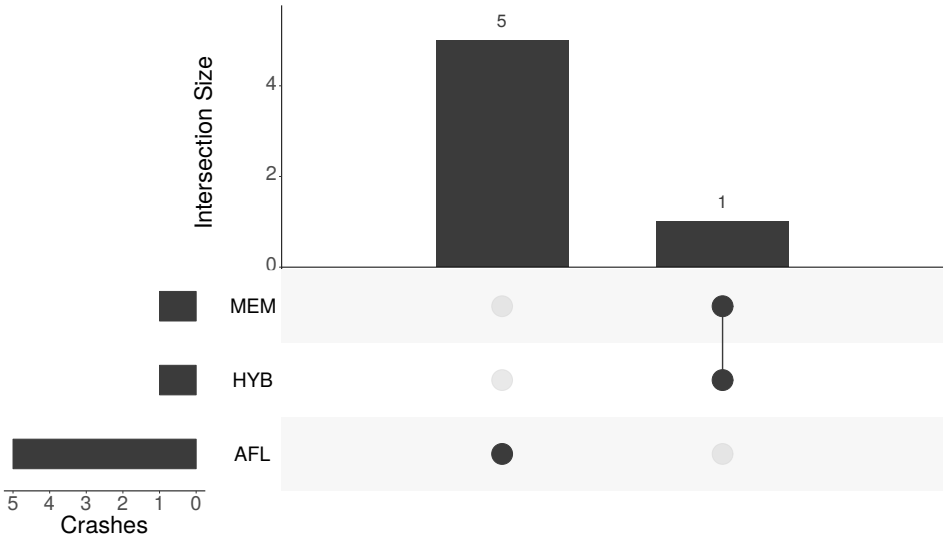
When using seed inputs without AddressSanitizer, AFL^S finds 5 distinct crashes whereas MEM^S and HYB^S do not find any. With AddressSanitizer (Figure 5.7), MEM+A^S and HYB+A^S perform as well as in the corresponding unseeded runs, whereas AFL+A^S finds fewer crashes than AFL+A. We attribute this to the use of a large seed input collection leading to longer individual runtimes and keeping the fuzzer from reaching more difficult crashes within the allotted time budget.

For `ImageMagick`, both with and without AddressSanitizer, MEM and HYB were able to find crashes not found by AFL, as shown in Figure 5.8. Unlike for `ffmpeg`, the crashes found by MEM and HYB in `ImageMagick` differ between the two fuzzers, with HYB outperforming MEM.

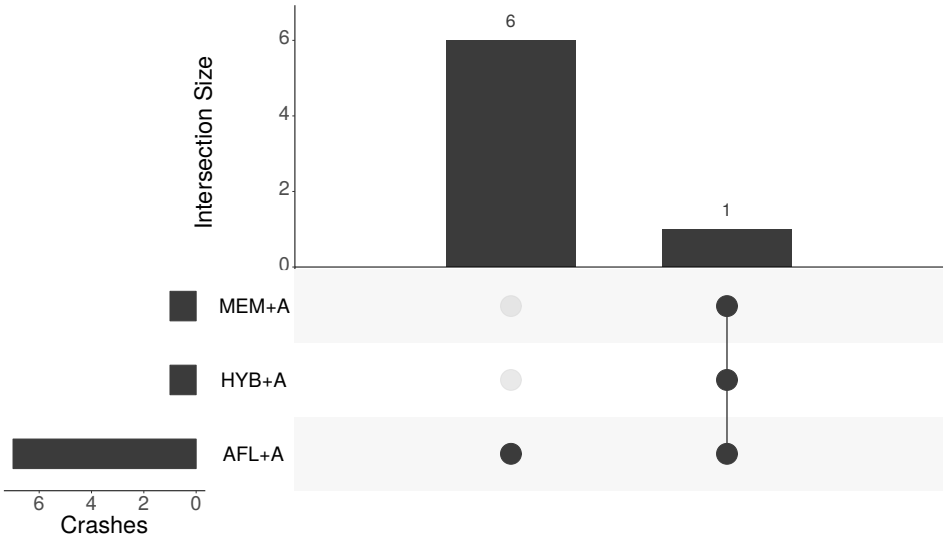
With seed inputs HYB^S and especially MEM^S perform well, finding 48 and 76 crashes not found by any other fuzzer, respectively (Section 5.4.2). Both exceed the performance of their unseeded counterparts whereas AFL^S does worse than AFL. With AddressSanitizer (Section 5.4.2), on the other hand, both MEM+A^S and HYB+A^S do worse than their unseeded counterparts, likely due to the increased runtime.

For `libxml2`, no fuzzing configuration found any crashes when not using AddressSanitizer. With AddressSanitizer, AFL+A found 149 distinct crashes whereas MEM+A and HYB+A did not find any crashes. MemFuzz performs better in the seeded experiments: As shown in Section 5.4.2, MEM^S finds 3 crashes that AFL^S does not. Moreover, when using AddressSanitizer, both MEM+A^S and HYB+A^S find crashes that AFL+A^S does not (Section 5.4.2).

Across all targets for which at least one fuzzing configuration found crashes, the number of crashes found by MEM and HYB is lower than AFL. We hypothesize that this is primarily due to the additional runtime overhead, as discussed in Section 5.4.3. However, despite finding a lower number of crashes overall, both MEM and HYB find



(a) Without AddressSanitizer



(b) With AddressSanitizer

Figure 5.6: `ffmpeg` crashes

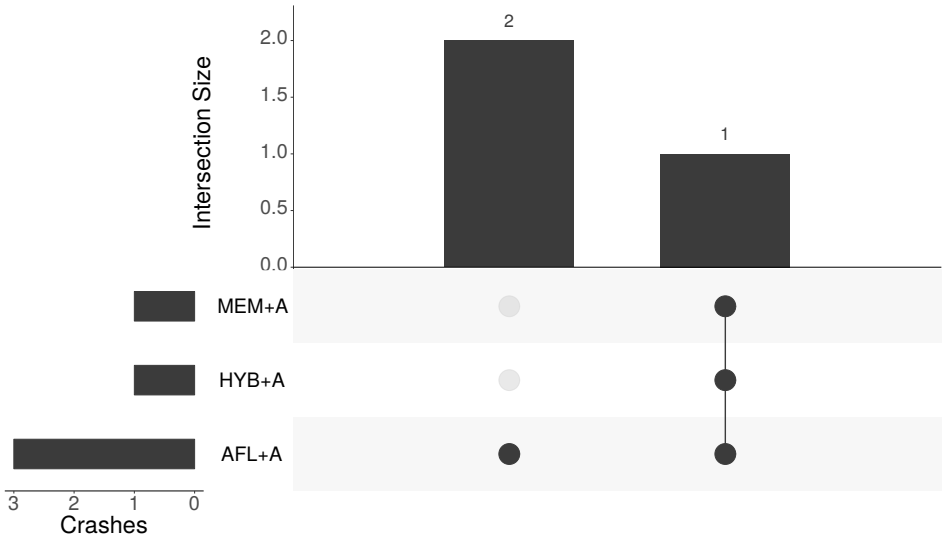


Figure 5.7: `ffmpeg` Crashes with AddressSanitizer (seeded)

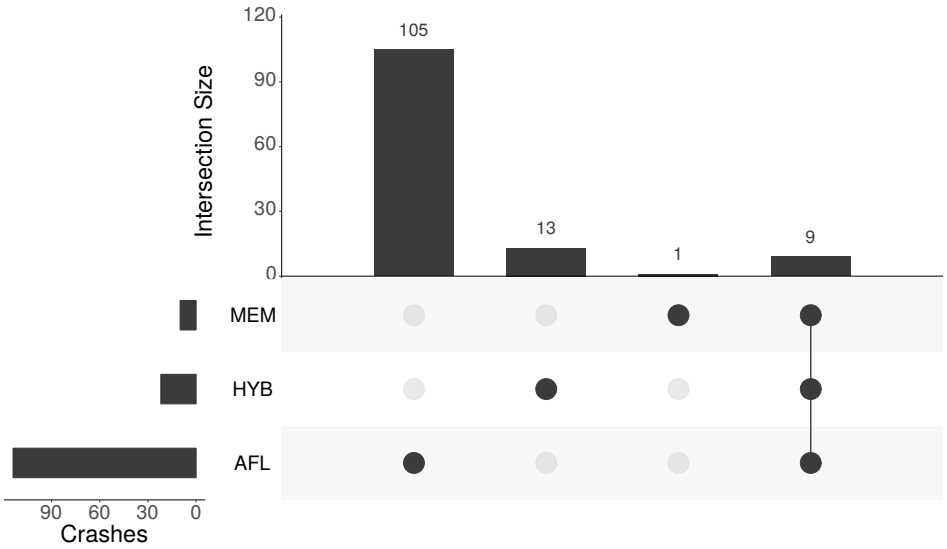
crashes not detected by AFL for all three targets we have evaluated, albeit not in all configurations. We therefore conclude that MemFuzz is capable of finding different crashes from conventional, coverage-guided evolutionary fuzzing.

5.4.3 RQ 2: Overhead

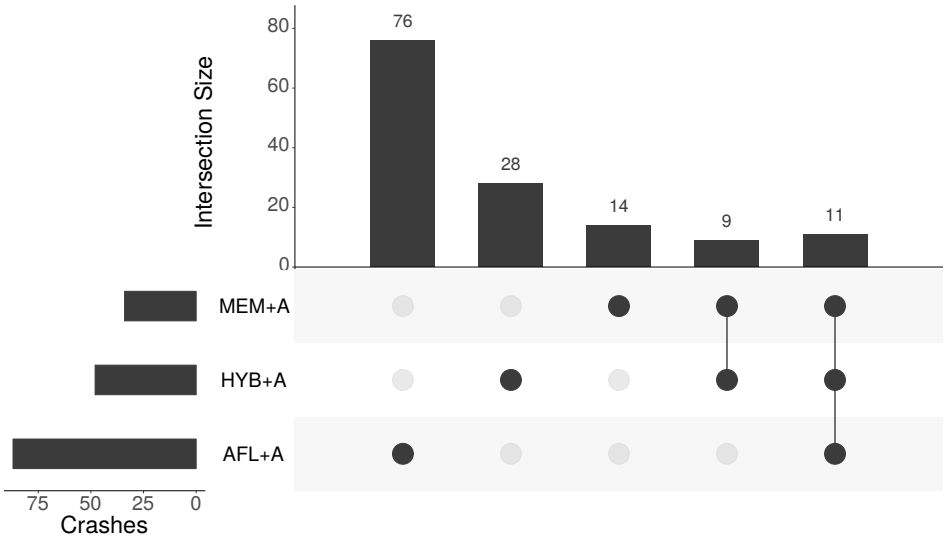
To determine the runtime overhead caused by the additional instrumentation and analysis required for MemFuzz, we compare the number of executions of each target configuration achieved by MEM and HYB to AFL.

For unseeded runs, the mean number of executions over five runs achieved by HYB and MEM relative to AFL is shown in Section 5.4.3. Without ASAN, MEM and HYB achieve between 11 % and 17 % of the executions achieved by AFL. With AddressSanitizer, they achieve between 25 % and 47 % of AFL+A executions. We hypothesize that this increase is due to the overhead incurred by AddressSanitizer instrumentation, which affects all fuzzers equally. In both cases, the overhead incurred by our MemFuzz implementation is highest for `ffmpeg`.

Overheads are lower for seeded runs, with MEM^S and HYB^S achieving 18 % and 17 % of AFL^S executions, respectively, on `ffmpeg`, an increase of 6 – 7 percentage points compared to the unseeded runs. On `ImageMagick`, the improvement is even larger at 27 – 28 percentage points, and on `libxml2`, performance relative to AFL^S is more than doubled compared to the unseeded runs. Configurations using AddressSanitizer see larger improvements still. On `ImageMagick`, MEM+A^S and HYB+A^S achieve 73 % of the executions of AFL+A^S. On `libxml2`, they achieve 64 % and 63 %, respectively. We hypothesize that the reduced slowdown for seeded runs

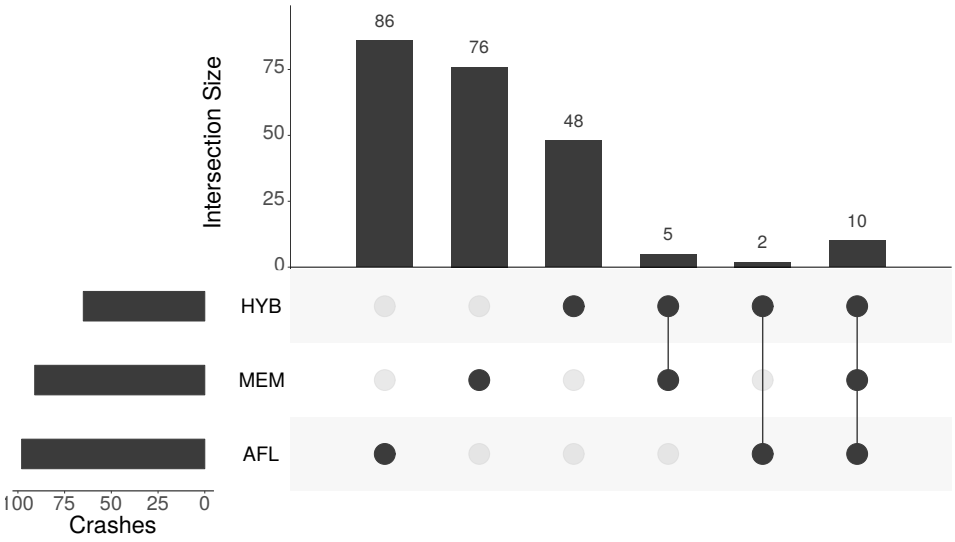


(a) Without AddressSanitizer

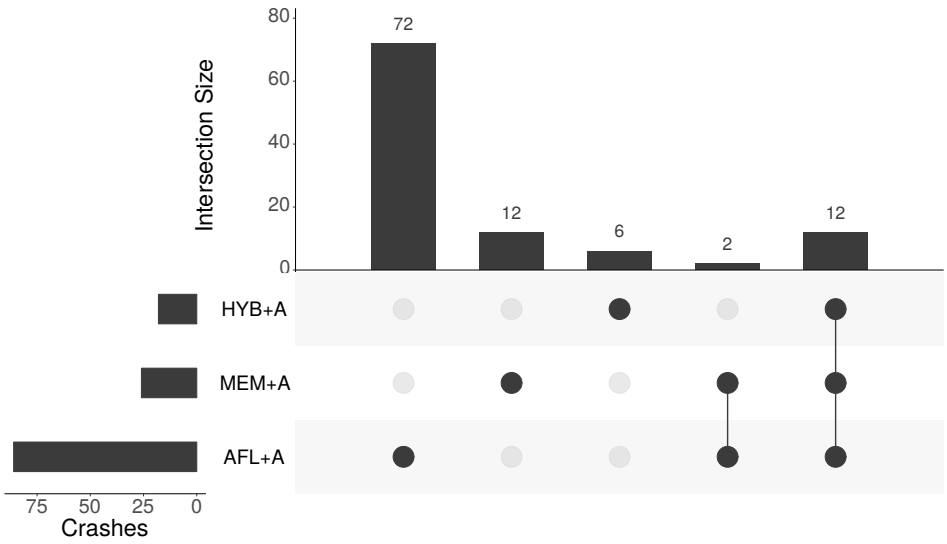


(b) With AddressSanitizer

Figure 5.8: `ImageMagick` crashes

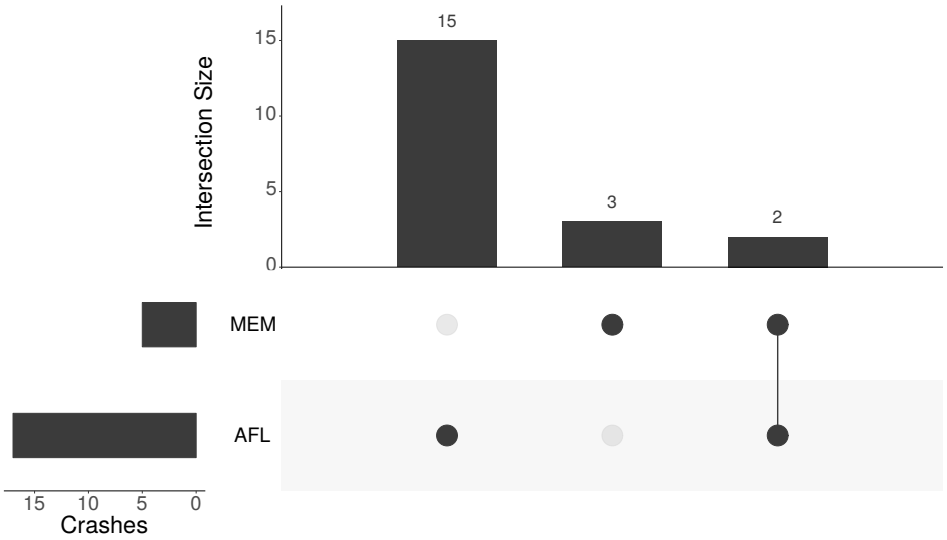


(a) Without AddressSanitizer

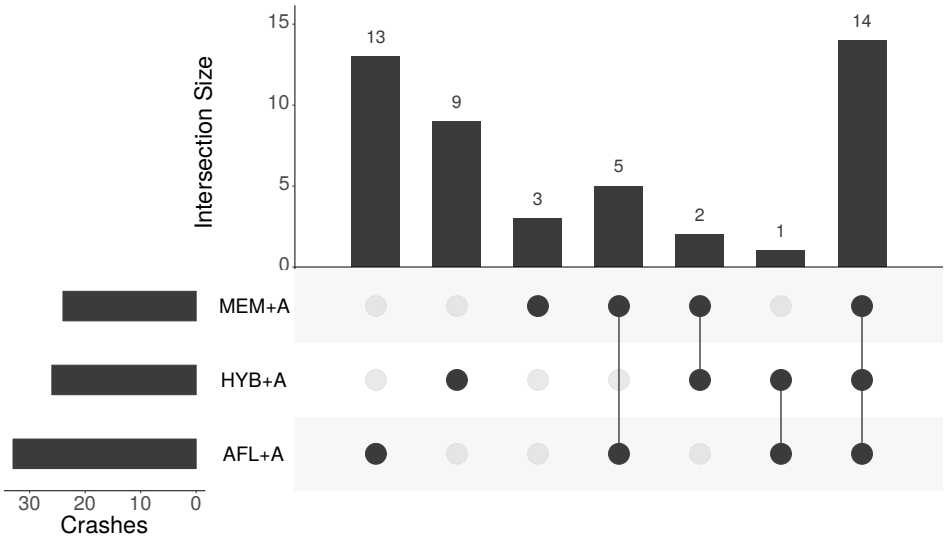


(b) With AddressSanitizer

Figure 5.9: `ImageMagick` crashes (seeded)

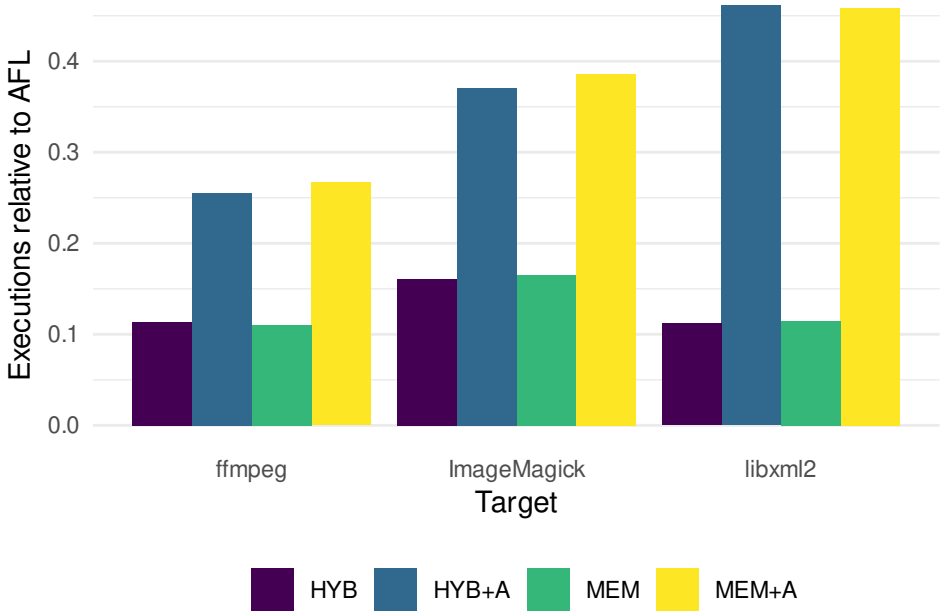


(a) Without AddressSanitizer

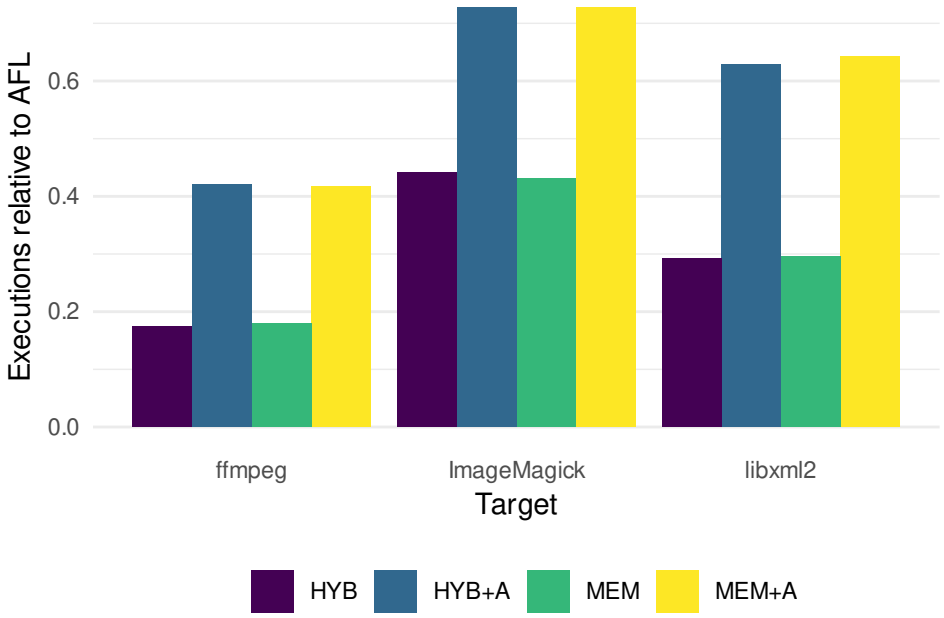


(b) With AddressSanitizer

Figure 5.10: `libxml2` crashes (seeded)



(a) Unseeded



(b) Seeded

Figure 5.11: Mean number of executions over five runs relative to AFL(+A).

is the result of longer execution durations for all fuzzers amortizing the more complex evaluation step after each execution.

We conclude that, while our implementation does incur substantial runtime overhead, the slowdown is highly variable between targets and configurations, ranging from 1.36× to more than 5×. The effect is lower when using appropriate seed inputs and AddressSanitizer, both of which are generally desirable in a wide variety of fuzzing use cases.

5.4.4 RQ 3: Coverage

As we modify the fuzzing engine and introduce additional instrumentation and consequently overhead (as discussed in Section 5.4.3), we expect our MemFuzz implementation to achieve a reduced edge coverage relative to conventional, coverage-guided evolutionary fuzzing. To assess the extent of the impact, we check the inputs generated by all fuzzers and compare the covered edges. Coverage for all fuzzers is checked on the same binary for each target. An edge is considered covered by a fuzzer if it was covered during at least one of the five executions by any of the eight parallel instances.

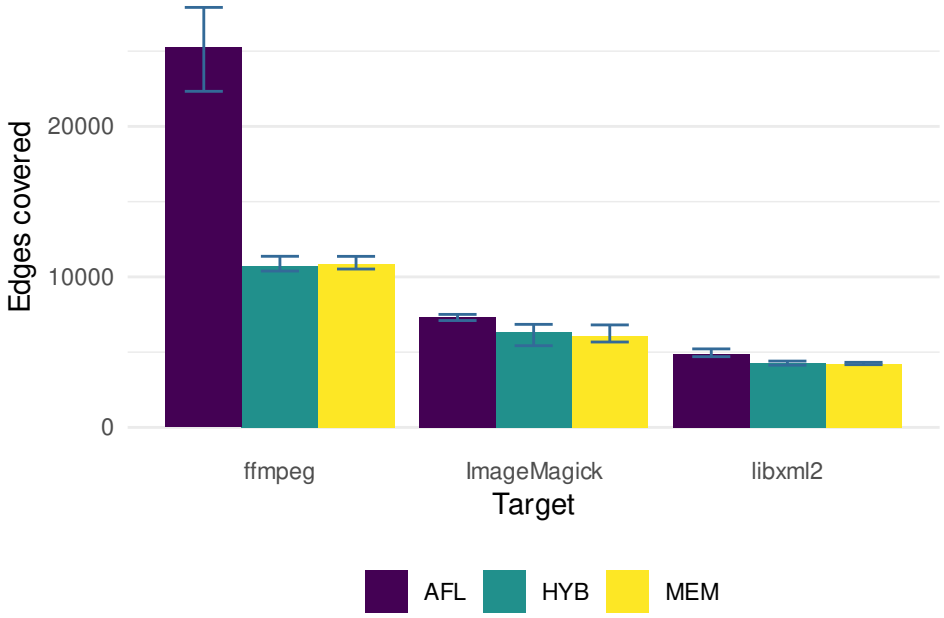
Results for the unseeded runs without AddressSanitizer are shown in Section 5.4.4. AFL consistently outperforms both MEM and HYB for all targets, but the size of the effect varies substantially between targets. On `ffmpeg`, AFL covers more than twice as many edges as MEM or HYB, whereas the difference on the other targets is much smaller, with both MEM and HYB achieving over 80 % of the coverage achieved by AFL. MEM and HYB perform similarly well on all three targets, with HYB having a slight edge on `ImageMagick` and `libxml2` while MEM does marginally better on `ffmpeg`.

For the seeded runs (Section 5.4.4), HYB and MEM are much more closely matched with AFL. On `ffmpeg`, both go from below 50 % of AFL's coverage to above 80 %. On `ImageMagick` and `libxml2`, MEM and HYB cover more than 96 % of the edges covered by AFL.

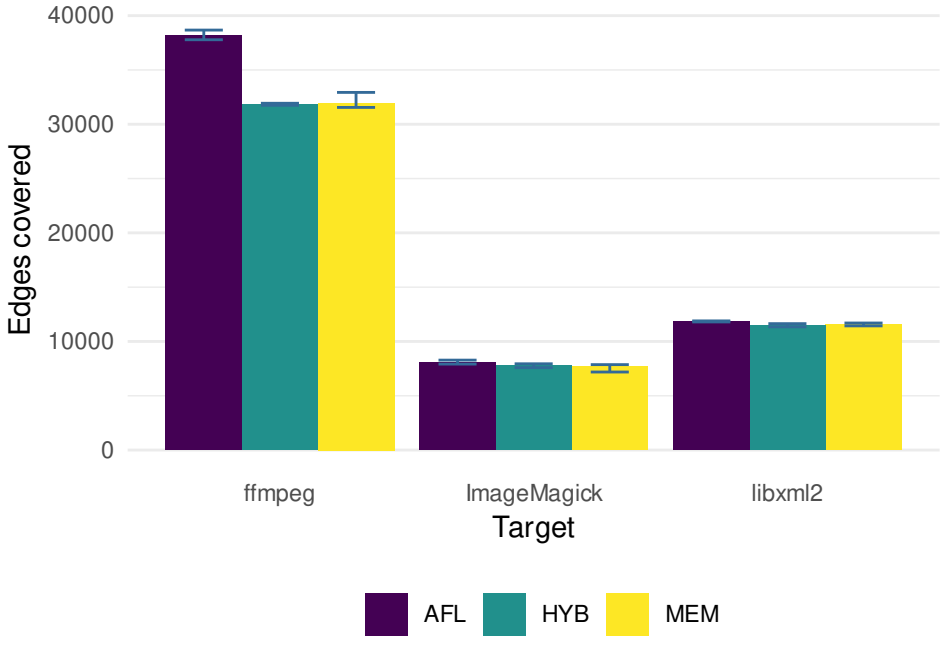
While some impact on coverage is to be expected due to the overheads reported in Section 5.4.3, our results show that the extent of this impact is highly dependent on the specific program under test and the quality of the seed inputs. We conclude that our approach does adversely affect edge coverage but the effect can be mitigated with careful seed selection.

5.4.5 RQ 4: Static Analysis

In Section 5.3.2, we describe our technique for reducing the number of instrumentation sites that we employ to reduce overall runtime overhead as each instrumentation point imposes a runtime cost. To assess how well our technique works in practice, we log both the number of instrumentation sites where instrumentation



(a) Unseeded



(b) Seeded

Figure 5.12: Mean edge coverage achieved by the different fuzzers over five runs. Error bars indicate minimum and maximum values.

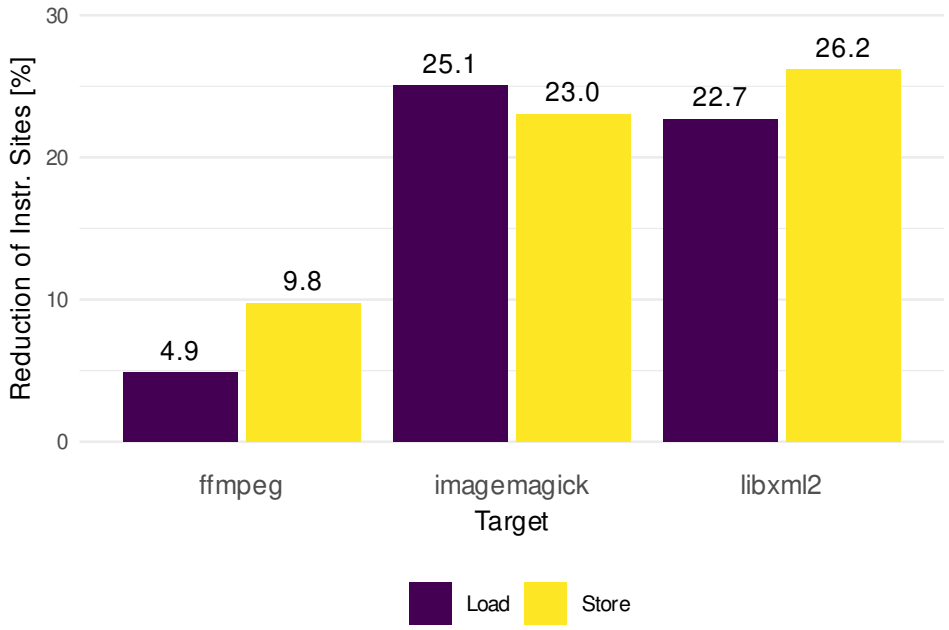


Figure 5.13: Reduction of Load/Store Instrumentation Sites Due to Static Analysis

Table 5.4: Number of Instrument Sites for each Evaluation Target

Application	Instrumentation Sites	Loads	Stores
`ffmpeg`	360 684	65.5 %	34.5 %
`ImageMagick`	295 373	66.4 %	33.6 %
`libxml2`	139 572	72.6 %	27.4 %

was applied and where it could be elided during the execution of our LLVM instrumentation pass. We report the percentage of `load` and `store` instructions in each of our fuzzing targets for which instrumentation could be elided by our technique in Figure 5.13.

For both `ImageMagick` and `libxml2` the number of necessary `load` instrumentations could be reduced by over 22 % and the number of `store` instrumentations by over 23 %. For `ffmpeg`, on the other hand, `load` instrumentations could be reduced by only about 5 % and `store` instrumentations by about 10 %. To put these percentages into perspective, Table 5.4 reports the absolute total numbers of instrumentation sites as well as how they are divided between loads and stores for each target. Overall, our technique saves 72 048 instrumentation points for `ImageMagick`, 33 016 for `libxml2`, and 23 681 for `ffmpeg`. As these numbers demonstrate, our static analysis and instrumentation elision technique is capable of significantly re-

ducing the number of necessary instrumentation points, which translates to a reduction of runtime overhead.

5.5 Discussion and Threats to Validity

In the previous section, we presented the experimental results obtained with our approach. In the following, we first discuss those results and potential future work in Section 5.5.1. This is followed by a discussion of the threats to validity of our results in Section 5.5.2.

5.5.1 Discussion

Here, we discuss the overhead incurred by our instrumentation and potential approaches for further reducing that overhead. We then discuss the relationship between overhead, coverage, and crash finding ability and briefly cover potential future work.

Instrumentation and Overhead

While the evaluation shows that our static analysis is capable of significantly reducing the number of instrumentation sites (cf. Section 5.4.5), our approach nonetheless incurs a substantial runtime overhead. Moreover, the evaluation shows that the target for which the static analysis was least effective at reducing the number of instrumentation points (`ffmpeg`) is not the target with the largest overhead. We hypothesize that this is due to our evaluation of the static analysis considering instrumentation sites at compile time, whereas runtime overhead is determined by the number of instrumentation sites encountered at runtime. Thus, to improve performance, it is more important to remove instrumentation sites in code that is frequently executed, rather than removing as many instrumentation sites as possible. While it may therefore be worthwhile to expend additional analysis effort to reduce instrumentation sites in frequently executed code paths, predicting where those paths will be at compile time is challenging. A more powerful static analysis could further reduce instrumentation sites throughout the target program and therefore also reduce runtime overhead.

Overhead, Coverage, and Crash Finding Ability

We find that the ability of our approach to find crashes is not directly linked to its overhead, the number of executions relative to AFL or the edge coverage relative to AFL. For instance, HYB+A and MEM+A perform better in terms of executions relative to AFL+A and coverage on `libxml2` than on the other two evaluation targets. At the same time, they do not find any crashes in `libxml2` when not using seed

inputs while AFL+A finds 149. Despite higher overheads and lower coverage, the performance in terms of crashes found relative to AFL+A is better on `ImageMagick` and `ffmpeg`. This suggests that our feedback mechanism is ill-suited for overcoming the early parsing stages in `libxml2`. When using seed inputs, this problem disappears, and consequently, MEM+A^S and HYB+A^S perform much better than their unseeded counterparts in terms of their ability to find crashes. These results highlight the importance of picking suitable seed inputs. Moreover, coverage does not appear to be a particularly good proxy for crash finding ability in general.

Future Work

There is potential to further reduce overheads through more powerful static analysis or a more optimized implementation. Another promising avenue for future work is the use of memory access instrumentation as part of the queue prioritization or for more targeted mutations in the fuzzer rather than just as an indicator for novelty. Exploring other strategies and combinations of memory access-guided and conventional coverage-guided fuzzing in a parallel fuzzing setting, such as running AFL, MEM, and HYB in parallel on a shared queue may also be worthwhile.

5.5.2 Threats to Validity

In this section, we discuss threats to the validity of our results due to our experiment setup, choice of evaluation targets, and crash bucketing approach.

Choice of Evaluation Targets

We have chosen three evaluation targets. All three targets are widely used and well-suited for fuzzing due to being written in a language that does not provide memory safety and handling complex data structures. Our results may not generalize to other targets, and to applications or libraries that do not have these properties.

Moreover, the number of instrumentation sites in a target application and the achievable reduction in that number depends not just on the target application but also on the employed compiler version and compiler optimizations. In other settings, the number or reduction in instrumentation sites may differ. Consequently, overhead may differ as well.

Crash Bucketing

To determine whether our proposed approach finds different crashes from conventional, coverage-guided evolutionary fuzzing, we use stack hashing to bucket the crashing inputs found by each fuzzer and assign a unique identifier to each bucket. However, crash bucketing using standard techniques is known to be imprecise [TKL18]. We first filter crashes using AFL's unique crashes heuristic as part

of the fuzzing process, then apply our stack hashing variant, but both techniques have known shortcomings and may suffer from both over- and underapproximation. Without manual inspection, it is not possible to precisely map crashes to underlying bugs in the general case. Therefore, it is possible that we over- or underapproximate the number of crashes. In the former case, we may erroneously map crashes with the same underlying cause to different buckets. In the latter case, crashes with different underlying causes get the same identifier. Both issues would affect both the total number of crashes reported as well as the comparisons between techniques performed in Section 5.4.2. As discussed in Section 5.4.2, we attempt to mitigate this concern by omitting specific memory addresses connected to crashes from the hash to ensure that deduplication does not grant an advantage to our proposed technique.

Experiment Durations

All results reported in our evaluation have been gathered with the setup and experiment duration described in Section 5.4.1. Substantially different experiment durations may affect how different fuzzers perform relative to each other. We have attempted to mitigate the effect of nondeterministic runtime behavior by repeating all experiments five times. We consider additionally assessing the impact of longer experiment durations to be infeasible as this would require an exceptionally large amount of computational resources.

5.6 Conclusion

We have introduced MemFuzz, an approach for using memory access instrumentation instead of or in addition to control flow information to guide evolutionary fuzzing. We have implemented a prototype of MemFuzz based on AFL. It comprises a static analysis and instrumentation component, a runtime component, and a modified version of the fuzzer capable of taking memory access information into account as a novelty criterion according to different, user-specified strategies. Our evaluation shows that, despite incurring non-negligible runtime overhead, the approach is capable of finding crashes that coverage-guided fuzzing does not find within the same time budget for all three targets in our evaluation. Our results demonstrate the feasibility of memory access instrumentation as a way to characterize program executions.

6 Summary and Conclusion

A wide variety of computing systems, ranging from low-power embedded systems in home electronics to powerful server systems, have become pervasive in our daily lives. As they have become more widespread, they have also become increasingly complex. We now rely on many computing systems to consistently and reliably provide important everyday functionality. That includes traditional embedded systems and safety-critical systems, but increasingly also IoT and smart home devices, smartphones, various cloud services, and other software systems built on general purpose, off-the-shelf components. Unlike conventional embedded systems, these systems undergo more frequent updates, commonly receive new functionality, and undergo less testing. This can result in shortcomings in dependability and security, which can have a substantial impact on users. On this background, dependability assessment and security testing for systems software is increasingly important. At the same time, due to the increasing complexity of modern software systems, conventional techniques for dependability assessment such as SFI suffer from long test execution latencies, which harms their adoption in practice. Complicating the issue further, many of the most important targets for such a dependability assessment are long-running systems, in which the lack of divergences after the execution of a specified workload is insufficient to determine whether an activated fault affected the system in a manner that could affect system behavior at a later time. Tackling this issue, in turn, exacerbates the SFI test execution latency problem.

In this thesis, we developed a technique to determine the impact of faulty components in monolithic operating systems, investigated ways to accelerate SFI testing, both for user space and kernel code, and, building on observations we made in using memory instrumentation for EPA, proposed a feedback-driven fuzzing technique that leverages information about input-dependent memory accesses as a guidance mechanism. The shared goal of these approaches and techniques is to improve dependability and security assessment techniques, especially with respect to their efficiency. We have developed these techniques with applicability to the lower levels of the software stack in mind and have tested them on software that can broadly be categorized as belonging to these lower levels. A substantial portion of the work described in this thesis deals with kernel code, which is crucial to the dependability of software systems. In summary, this thesis has investigated the following research questions and, in the process, made the following contributions:

Research Question 1 (RQ 1): How can the effects of faulty OS components on other parts of the system be identified in the absence of externally visible failures?

Modern OS kernels consist of many interacting components. Even in monolithic designs such as the Linux kernel, the system is divided into components, albeit without runtime isolation between them. Some components implement core functionality, others are tailored for specific tasks, configurations, or use cases. In monolithic designs, the lack of runtime isolation makes it difficult to assess the effect of faults in one component on other parts of the system. In such systems, faulty code can directly alter the state of other components or provide invalid values to them in ways that are challenging to detect. Since OS kernels are long-running systems, the absence of a system failure during the execution of a test workload is insufficient to rule out effects that can affect the system's behavior at a later time. Therefore, SFI tests for kernel code can benefit from a way to detect state corruption due to activated faults.

Contribution 1 (C 1): A tracing-based approach for assessing the effects of faults in kernel modules.

In Chapter 2, we presented TrEKer, our approach for assessing the effects of faults in kernel modules on other parts of the system in monolithic OS kernels. Like previous work that has attempted to tackle similar issues, we make use of execution tracing. Due to the shared address space and lack of runtime isolation, any memory access in the kernel is a potential instance of component interaction. Therefore, fine-grained tracing on the granularity of individual memory accesses is needed. As we target kernel code, we cannot rely on existing user space tracing solutions. We restrict our instrumentation to the SFI target component and insert instrumentation code at compile time. We develop a technique to process and analyze the gathered traces to determine which memory accesses constitute potential component interaction by building up a graph structure and checking which addresses are reachable from memory addresses known to components besides the injection target. To tackle non-deterministic runtime behavior of our target system due to, for instance, scheduling effects, we develop a trace comparison approach that can compare the execution trace of a faulty version to a merged trace of an arbitrary number of golden runs. We demonstrate the applicability of our approach by using it to trace the effects of software faults injected into three widely used Linux kernel components. We find that conventional oracles based on externally visible system behavior would misclassify up to approximately 10 % of seemingly successful executions. Our approach achieves a false positive rate below 1 %.

Research Question 2 (RQ 2): How can SFI experiments be accelerated and adapted to efficiently utilize modern, parallel hardware?

Due to the growing complexity of modern software systems, the number of faulty versions that need to be executed in SFI testing is increasing. This leads to increasing SFI test latencies and can render comprehensive SFI testing of complex software systems infeasible in some scenarios. This is particularly problematic for systems software, where comprehensive dependability assessment is especially important as the dependability of systems software is crucial for overall system dependability, but the basic problem arises throughout the software stack. To mitigate the problem of long SFI test latencies, approaches to speed up the execution of individual SFI tests, run more SFI tests in parallel to take advantage of modern, parallel hardware, or reduce the number of required SFI tests are necessary. At the same time, it is crucial that such approaches do not adversely affect result validity, for instance, by allowing for interference between different faulty versions due to shared resources or increased system load, or by leaving out faulty versions that could have revealed relevant failures.

Contribution 2 (C 2): A technique for accelerating SFI experiments by avoiding redundant work and facilitating parallelization.

In Chapter 3, we described a technique to accelerate SFI experiments for user mode code. Our technique, called FastFI, accelerates SFI testing in three different ways: It reduces the amount of faulty versions that are executed in which the fault is not activated by the given workload, it avoids repeatedly re-executing common execution prefixes that are shared between different faulty versions, and it facilitates SFI test parallelization to take advantage of modern parallel hardware. Furthermore, FastFI reduces build times by reducing the amount of redundant re-compilation. Our approach uses static analysis to group different faulty versions together at function granularity and generate a single binary including all generated faulty versions as well as the required control logic for SFI experiments. Since our approach relies on OS-provided process management functionality, it applies to user mode code and cannot be applied to kernel code. We rely on process isolation to prevent different faulty versions from interfering with each other. We implement a FastFI prototype that targets user mode software written in C and apply that prototype to four applications from the PARSEC benchmark suite. We achieve sequential speedups of up to 3.6×, parallel speedups of up to 20.6× with 16 parallel instances, reductions of up to 52.1 % in the number of executed faulty versions, and build time speedups of up to 13.7× without adversely affecting result validity.

Research Question 3 (RQ 3): How can SFI and EPA on OS components be accelerated and efficiently parallelized?

Dependability assessments of OS kernel code is particularly challenging, especially in monolithic kernel designs. Monolithic systems lack well-defined component boundaries, interfaces, or runtime isolation. Therefore faults in individual components can affect other parts of the system in an arbitrary manner, for instance, by directly altering the state of other components or by providing invalid values to other components either explicitly, through arguments or return values, or implicitly, through shared memory communication. Since OS kernels are long-running systems, relying on SFI oracles to detect externally visible system failures is insufficient. Additional instrumentation to detect instances of state corruption using EPA as described in Chapter 2 is required, which in turn further increases test latencies. Accelerating SFI tests of OS kernel code is not straightforward, particularly in the presence of instrumentation for the purpose of error propagation analysis, without affecting result validity.

Contribution 3 (C 3): An approach to reduce SFI test latencies for OS kernel components while allowing detection of internal state corruption.

In Chapter 4, we described an approach for accelerating SFI tests of OS kernel code with TrEKer-based EPA instrumentation. Our approach is conceptually related to the approach we described in Chapter 3, FastFI, in that it also tackles the common prefix problem, avoids executing inactive faulty versions at function granularity, facilitates parallelization, and reduces build times. Our approach targets SFI testing of kernel code in virtualized environments. We integrate all faulty versions of a kernel module into a single module, which is then instrumented to provide the fine-grained tracing required for error propagation analysis. The integrated kernel module interfaces with a separate runtime module that implements logging and handles communication with the SFI experiment control logic running on the host. The experiment control logic is responsible for scheduling and monitoring the execution of faulty versions as well as experiment result logging. To realize fast VM cloning, we make use of modern file system and VMM features including VM snapshotting and CoW file systems. We enhance TrEKer, the EPA approach presented in Chapter 2, to handle the fragmented traces generated by our integrated execution model. To demonstrate the applicability of our approach, we apply it to seven widely used Linux file systems. We achieve sequential SFI test speedups of up to 2.45×, parallel speedups of up to 36.8× using 16 parallel instances relative to sequential execution in the conventional mode, and build time speedups of up to 100.4× with instrumentation and 256.7× without. Our approach does not adversely affect SFI result validity.

Research Question 4 (RQ 4): Can selective instrumentation of memory accesses characterize program executions in a manner suitable to guide feedback-driven evolutionary fuzzing?

Fuzzing encompasses a number of testing techniques that are widely used for finding security and dependability issues. Approaches that use information about previous executions of the SUT to pick which inputs to mutate further are termed feedback-driven fuzzing. Most commonly the feedback mechanism used is a form of structural code coverage such as edge or basic block coverage. However, as described in Chapter 2, our work on memory access tracing for kernel EPA shows that memory accesses performed by the SUT can be a meaningful way to characterize a program execution. This approach may provide more fine-grained information than control flow can provide. However, using memory access traces to guide fuzzing is raises several challenges. Overhead must be minimized as performance is critical, steps must be taken to ensure that the instrumentation and tracing do not affect the memory addresses used in the SUT, and input-independent memory accesses should be excluded from instrumentation.

Contribution 4 (C 4): A technique to use memory access instrumentation to guide evolutionary fuzzing.

In Chapter 5, we described our approach, called MemFuzz, that makes use of information about input-dependent memory accesses to guide a feedback-driven fuzzer. We use conservative, intraprocedural static analysis to filter out input-independent memory accesses. At runtime, memory addresses — but not values — used by the SUT are stored in a bloom filter and provided to the fuzzer as a feedback mechanism. Since the bloom filter is a fixed size data structure, MemFuzz does not need to allocate or free memory at runtime and therefore does not affect the memory addresses used by the SUT. We implement a MemFuzz prototype based on the AFL fuzzer, which is a widely used feedback-driven fuzzer which normally relies on edge coverage. For the static analysis required to filter out input-independent memory accesses and the memory access instrumentation, we base our implementation on the LLVM-based instrumentation pass that ships with AFL. We apply our prototype to three widely used target programs in several different configurations, including running the SUT with and without additional AddressSanitizer instrumentation and fuzzing it with and without seed inputs. We find that, for each target program, MemFuzz detects distinct crashes from those found by AFL in at least one configuration. We conclude that different ways of characterizing program executions beyond control flow coverage can be used to guide fuzzers to find different and distinct crashes.

As we grow increasingly dependent on complex software systems in everyday live, it is essential that these systems are dependable and secure. To ensure that this is the case, dependability assessment and security testing techniques are needed, and numerous such techniques have been developed. However, for some of the most important target systems, commonly used techniques such as SFI are challenging to apply due to the extraordinary test latencies they incur. In this thesis, we have investigated techniques related to fault removal and fault forecasting, developed improved SFI test oracles for OS kernel code, accelerated SFI testing of user mode and kernel code, and developed a fuzzing technique leveraging memory access information as a guidance mechanism. Approaches capable of finding dependability and robustness issues efficiently are crucial for dealing with increasingly complex software systems providing increasingly important functionality. The approaches developed in this thesis facilitate the dependability assessment of such systems.

List of Figures

List of Tables

Bibliography

[Aid+01] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson. "GOOFI: Generic Object-Oriented Fault Injection Tool". In: *2001 International Conference on Dependable Systems and Networks*. 2001, pp. 83–88. DOI: 10.1109/DSN.2001.941394.

[And72] James P. Anderson. "Information Security in a Multi-User Computer Environment". In: vol. 12. Advances in Computers. Elsevier, 1972, pp. 1–36. DOI: https://doi.org/10.1016/S0065-2458(08)60506-9.

[APB14] M. R. Aliabadi, K. Pattabiraman, and N. Bidokhti. "Soft-LLFI: A Comprehensive Framework for Software Fault Injection". In: *Proceedings of International Symposium on Software Reliability Engineering Workshops*. 2014, pp. 1–5.

[Arl+02] J. Arlat, J.C. Fabre, M. Rodríguez, and F. Salles. "Dependability of COTS Microkernel-Based Systems". In: *IEEE Transactions on Computers* 51.2 (2002), pp. 138–163. DOI: 10.1109/12.980005.

[Avi+04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. "Basic concepts and taxonomy of dependable and secure computing". In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (Jan. 2004), pp. 11–33. DOI: 10.1109/TDSC.2004.2.

[Ban+10] Takayuki Banzai, Hitoshi Koizumi, Ryo Kanbayashi, Takayuki Imada, Toshihiro Hanawa, and Mitsuhisa Sato. "D-Cloud: Design of a Software Testing Environment for Reliable Distributed Systems Using Cloud Computing Technology". In: *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. 2010, pp. 631–636. DOI: 10.1109/CCGRID.2010.72.

[BC12] Radu Banabic and George Candea. "Fast black-box testing of system recovery code". In: *Proceedings of the 7th ACM european conference on Computer Systems*. EuroSys'12. 2012, pp. 281–294. DOI: 10.1145/2168836.2168865.

[Bel+15] Jonathan Bell, Gail Kaiser, Eric Melski, and Mohan Dattatreya. "Efficient dependency detection for safe Java test acceleration". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE'15. New York, New York, USA: ACM Press, 2015, pp. 770–781. DOI: 10.1145/2786805.2786823.

[Bel17] Fabrice Bellard. *QEMU*. 2017. URL: https://www.qemu.org.

[BGA03] D. Bruening, T. Garnett, and S. Amarasinghe. "An infrastructure for adaptive dynamic optimization". In: *International Symposium on Code Generation and Optimization*. CGO '03. 2003, pp. 265–275.

[Bie11] Christian Bienia. "Benchmarking Modern Multiprocessors". PhD thesis. Princeton University, Jan. 2011.

[BK14] Jonathan Bell and Gail Kaiser. "Unit Test Virtualization with VMVM". In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 550–561. DOI: 10.1145/2568225.2568248.

[BL07] Prashanth P. Bungale and Chi-Keung Luk. "PinOS: A Programmable Framework for Whole-system Dynamic Instrumentation". In: *Proceedings of the 3rd International Conference on Virtual Execution Environments*. 2007, pp. 137–147. DOI: 10.1145/1254810.1254830.

[Böh+17] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. "Directed Greybox Fuzzing". In: *2017 ACM Conference on Computer and Communications Security*. 2017, pp. 2329–2344.

[BPR18] M. Böhme, V. Pham, and A. Roychoudhury. "Coverage-based Greybox Fuzzing as Markov Chain". In: *IEEE Transactions on Software Engineering* (2018).

[Bro18] Eric Brown. *A Closer Look at Voice-Assisted Speakers*. Nov. 2018. URL: https://www.linux.com/tutorials/closer-look-voice-assisted-speakers (visited on 03/05/2020).

[Bud+80] Timothy A. Budd, Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. "Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of Programs". In: *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*. 1980, pp. 220–233.

[CB89] R. Chillarege and N. Bowen. "Understanding large system failures-a fault injection experiment". In: *[1989] The Nineteenth International Symposium on Fault-Tolerant Computing. Digest of Papers*. 1989, pp. 356–363. DOI: 10.1109/FTCS.1989.105592.

[CBZ10] George Candea, Stefan Bucur, and Cristian Zamfir. "Automated Software Testing as a Service". In: *Proceedings of the 1st ACM symposium on Cloud computing*. SOCC'10. 2010, pp. 155–160. DOI: 10.1145/1807128.1807153.

[CC18] P. Chen and H. Chen. "Angora: Efficient Fuzzing by Principled Search". In: *IEEE Security & Privacy*. 2018, pp. 711–725.

[Cio+10] Liviu Ciortea, Cristian Zamfir, Stefan Bucur, Vitaly Chipounov, and George Candea. "Cloud9: A Software Testing Service". In: *SIGOPS Operating Systems Review* 43.4 (Jan. 2010), pp. 5–10.

[Cis20] Cisco. *Cisco Annual Internet Report (2018 – 2023) White Paper*. Feb. 2020. URL: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html (visited on 03/05/2020).

[CLG17] Jake R Conway, Alexander Lex, and Nils Gehlenborg. "UpSetR: an R package for the visualization of intersecting sets and their properties". In: *Bioinformatics* 33.18 (2017), pp. 2938–2940.

[CMd17] Jeanderson Candido, Luis Melo, and Marcelo d'Amorim. "Test Suite Parallelization in Open-source Projects: A Study on Its Usage and Impact". In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. ASE 2017. Urbana-Champaign, IL, USA: IEEE Press, 2017, pp. 838–848.

[CMS98] J. Carreira, H. Madeira, and J. G. Silva. "Xception: a technique for the experimental evaluation of dependability in modern computers". In: *IEEE Transactions on Software Engineering* 24.2 (Feb. 1998), pp. 125–136. DOI: 10.1109/32.666826.

[CN13] D. Cotroneo and R. Natella. "Fault Injection for Software Certification". In: *IEEE Security & Privacy* 11.4 (2013), pp. 38–45. DOI: 10.1109/MSP.2013.54.

[Col] Yann Collet. *xxHash*. URL: http://xxhash.com (visited on 03/06/2020).

[Cop+17] Nicolas Coppik, Oliver Schwahn, Stefan Winter, and Neeraj Suri. "TrEKer: Tracing Error Propagation in Operating System Kernels". In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. ASE 2017. Urbana-Champaign, IL, USA: IEEE Press, 2017, pp. 377–387. DOI: 10.1109/ASE.2017.8115650.

[Cot+13] D. Cotroneo, D. Di Leo, F. Fucci, and R. Natella. "SABRINE: State-based robustness testing of operating systems". In: *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. 2013, pp. 125–135. DOI: 10.1109/ASE.2013.6693073.

[Cot+15] D. Cotroneo, L. De Simone, F. Fucci, and R. Natella. "MoIO: Run-time monitoring for I/O protocol violations in storage device drivers". In: *Proceedings of the 26th IEEE International Symposium on Software Reliability Engineering*. 2015, pp. 472–483. DOI: 10.1109/ISSRE.2015.7381840.

[CP95] Jeffrey A Clark and Dhiraj K Pradhan. "Fault injection: A method for validating computer-system dependability". In: *Computer* 28.6 (1995), pp. 47–56.

[CSS19] Nicolas Coppik, Oliver Schwahn, and Neeraj Suri. "MemFuzz: Using Memory Accesses to Guide Fuzzing". In: *12th IEEE International Conference on Software Testing, Verification and Validation*. ICST 2019. Xi'an, China, Apr. 2019, pp. 48–58. DOI: 10.1109/ICST.2019.00015.

[CSS20] Nicolas Coppik, Oliver Schwahn, and Neeraj Suri. "Fast Kernel Error Propagation Analysis in Virtualized Environments". In: *14th USENIX Symposium on Operating Systems Design and Implementation*. OSDI 2020. 2020. [under submission].

[DD06] Mathieu Desnoyers and Michel R Dagenais. "The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux". In: *Ottawa Linux Symposium*. 2006, pp. 209–224.

[Di +12] Domenico Di Leo, Fatemeh Ayatolahi, Behrooz Sangchoolie, Johan Karlsson, and Roger Johansson. "On the Impact of Hardware Faults–An Investigation of the Relationship between Workload Inputs and Failure Mode Distributions". In: *Proceedings of the 31st International Conference on Computer Safety, Reliability, and Security*. SAFECOMP'12. 2012, pp. 198–209. DOI: 10.1007/978-3-642-33678-2_17.

[DM06] J. A. Duraes and H. S. Madeira. "Emulation of Software faults: A Field Data Study and a Practical Approach". In: *IEEE Transactions on Software Engineering* 32.11 (2006), pp. 849–867. DOI: 10.1109/TSE.2006.113.

[DO91] R. A. DeMillo and A. J. Offutt. "Constraint-based automatic test data generation". In: *IEEE Transactions on Software Engineering* 17.9 (Sept. 1991), pp. 900–910.

[Dua+06] Alexandre Duarte, Walfredo Cirne, Francisco Brasileiro, and Patricia Machado. "GridUnit: Software Testing on the Grid". In: *Proceedings of the 28th International Conference on Software Engineering*. ICSE '06. New York, NY, USA: ACM, 2006, pp. 779–782. DOI: 10.1145/1134285.1134410.

[FBG12] Peter Feiner, Angela Demke Brown, and Ashvin Goel. "Comprehensive Kernel Instrumentation via Dynamic Binary Translation". In: *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*. 2012, pp. 135–146. DOI: 10.1145/2150976.2150992.

[FFm] FFmpeg. *FFmpeg Automated Testing Environment*. URL: https://www.ffmpeg.org/fate.html (visited on 03/31/2020).

[FX02] C. Fetzer and Zhen Xiao. "An automated approach to increasing the robustness of C libraries". In: *2002 International Conference on Dependable Systems and Networks*. 2002, pp. 155–164. DOI: 10.1109/DSN.2002.1028896.

[Gam+17] Alessio Gambi, Sebastian Kappler, Johannes Lampel, and Andreas Zeller. "CUT: Automatic Unit Testing in the Cloud". In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2017. New York, NY, USA: ACM, 2017, pp. 364–367. DOI: 10.1145/3092703.3098222.

[Gan+18] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen. "CollAFL: Path Sensitive Fuzzing". In: *IEEE Security & Privacy*. 2018, pp. 679–696.

[Gro+16] Alex Groce, Mohammad Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. "Cause reduction: delta debugging, even without bugs". In: *Software Testing, Verification and Reliability* 26.1 (2016), pp. 40–68.

[Gun+11] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. "FATE and DESTINI: A Framework for Cloud Recovery Testing". In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. NSDI'11. 2011, pp. 238–252.

[Han+10] T. Hanawa, T. Banzai, H. Koizumi, R. Kanbayashi, T. Imada, and M. Sato. "Large-Scale Software Testing Environment Using Cloud Computing Technology for Dependable Parallel and Distributed Systems". In: *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*. 2010, pp. 428–433. DOI: 10.1109/ICSTW.2010.59.

[Hen+14] Andrew Henderson, Aravind Prakash, Lok Kwong Yan, Xunchao Hu, Xujiewen Wang, Rundong Zhou, and Heng Yin. "Make It Work, Make It Right, Make It Fast: Building a Platform-neutral Whole-system Dynamic Binary Analysis Platform". In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 2014, pp. 248–258. DOI: 10.1145/2610384.2610407.

[Hen+16] A. Henderson, L. Yan, X. Hu, A. Prakash, H. Yin, and S. McCamant. "DECAF: A Platform-Neutral Whole-System Dynamic Binary Analysis Platform". In: *IEEE Transactions on Software Engineering* PP.99 (2016), pp. 1–1. DOI: 10.1109/TSE.2016.2589242.

[Hsu+18] Chin-Chia Hsu, Che-Yu Wu, Hsu-Chun Hsiao, and Shih-Kun Huang. "INSTRIM: Lightweight Instrumentation for Coverage-guided Fuzzing". In: *Workshop on Binary Analysis Research* (2018).

[HTI97] Mei-Chen Hsueh, T. K. Tsai, and R. K. Iyer. "Fault Injection Techniques and Tools". In: *Computer* 30.4 (Apr. 1997), pp. 75–82.

[IDC20] IDC. *Smartphone Market Share – OS*. 2020. URL: https://www.idc.com/promo/smartphone-market-share/os (visited on 03/05/2020).

[IEE18] IEEE and The Open Group. "IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7". In: *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)* (Jan. 2018), pp. 1–3951. DOI: 10.1109/IEEESTD.2018.8277153.

[INR18] INRIA. *Coccinelle Website*. 2018. URL: http://coccinelle.lip6.fr (visited on 03/30/2020).

[Int10] International Electrotechnical Commission. *IEC 61508: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems*. 2010.

[Int11] International Organization for Standardization. *ISO 26262: Road Vehicles – Functional Safety*. 2011.

[IVD15] I. Irrera, M. Vieira, and J. Duraes. "Adaptive Failure Prediction for Computer Systems: A Framework and a Case Study". In: *IEEE 16th International Symposium on High Assurance Systems Engineering*. 2015, pp. 142–149. DOI: 10.1109/HASE.2015.29.

[JGS11] Pallavi Joshi, Haryadi S. Gunawi, and Koushik Sen. "PREFAIL: A Programmable Tool for Multiple-failure Injection". In: *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*. 2011, pp. 171–188. DOI: 10.1145/2048066.2048082.

[JH08] Yue Jia and M. Harman. "Constructing Subtle Faults Using Higher Order Mutation Testing". In: *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*. Sept. 2008, pp. 249–258. DOI: 10.1109/SCAM.2008.36.

[JH11a] Y. Jia and M. Harman. "An Analysis and Survey of the Development of Mutation Testing". In: *IEEE Transactions on Software Engineering* 37.5 (Sept. 2011), pp. 649–678. DOI: 10.1109/TSE.2010.62.

[JH11b] Y. Jia and M. Harman. "An Analysis and Survey of the Development of Mutation Testing". In: *IEEE Transactions on Software Engineering* 37.5 (Sept. 2011), pp. 649–678.

[JLC18] Bo Jiang, Ye Liu, and W. K. Chan. "ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection". In: *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering*. 2018, pp. 259–269.

[JZ08] X. Ju and H. Zou. "Operating System Robustness Forecast and Selection". In: *Proceedings of the 19th IEEE International Symposium on Software Reliability Engineering*. 2008, pp. 107–116. DOI: 10.1109/ISSRE.2008.10.

[Kap01] Gregory M Kapfhammer. "Automatically and Transparently Distributing the Execution of Regression Test Suites". In: *Proceedings of the 18th International Conference on Testing Computer Software*. 2001.

[KB13] Piyus Kedia and Sorav Bansal. "Fast Dynamic Binary Translation for the Kernel". In: *ACM Symposium on Operating Systems Principles*. 2013, pp. 101–115. DOI: 10.1145/2517349.2522718.

[KD00] P. Koopman and J. DeVale. "The Exception Handling Effectiveness of POSIX Operating Systems". In: *IEEE Transactions on Software Engineering* 26.9 (2000), pp. 837–848. DOI: 10.1109/32.877845.

[KIT93a] W. I. Kao, R. K. Iyer, and D. Tang. "FINE: A fault injection and monitoring environment for tracing the UNIX system behavior under faults". In: *IEEE Transactions on Software Engineering* 19.11 (Nov. 1993), pp. 1105–1118. DOI: 10.1109/32.256857.

[KIT93b] W. I. Kao, R. K. Iyer, and D. Tang. "FINE: A fault injection and monitoring environment for tracing the UNIX system behavior under faults". In: *IEEE Transactions on Software Engineering* 19.11 (Nov. 1993), pp. 1105–1118.

[KKS98] N. P. Kropp, P. J. Koopman, and D. P. Siewiorek. "Automated robustness testing of off-the-shelf software components". In: *Proceedings of the Twenty-Eigth Annual International Symposium on Fault-Tolerant Computing*. 1998, pp. 230–239. DOI: 10.1109/FTCS.1998.689474.

[Kni16] Kate Knibbs. *Nest Thermostats Are Having Battery Problems and There's No Fix Yet*. Jan. 2016. URL: https://gizmodo.com/nest-thermostats-are-having-battery-problems-and-theres-1751800309 (visited on 03/05/2020).

[Koo+97] P. Koopman, J. Sung, C. Dingman, D. Siewiorek, and T. Marz. "Comparing operating systems using robustness benchmarks". In: *Proceedings of the Sixteenth Symposium on Reliable Distributed Systems*. 1997, pp. 72–79. DOI: 10.1109/RELDIS.1997.632800.

[LA04] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. CGO '04. Palo Alto, California: IEEE Computer Society, 2004.

[Lag+09] Horacio Andrés Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M. Rumble, Eyal de Lara, Michael Brudno, and Mahadev Satyanarayanan. "SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing". In: *Proceedings of the 4th ACM European Conference on Computer Systems*. EuroSys '09. Nuremberg, Germany: ACM, 2009, pp. 1–12. DOI: 10.1145/1519065.1519067.

[Lan+14] A. Lanzaro, R. Natella, S. Winter, D. Cotroneo, and N. Suri. "An Empirical Study of Injected versus Actual Interface Errors". In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 2014, pp. 397–408.

[Las05] Alexey Lastovetsky. "Parallel testing of distributed software". In: *Information and Software Technology* 47.10 (2005), pp. 657–662.

[Lem+18] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. "PerfFuzz: Automatically Generating Pathological Inputs". In: *Proceedings of the 2018 International Symposium on Software Testing and Analysis*. 2018, pp. 254–265.

[Lex+14] A. Lex, N. Gehlenborg, H. Strobelt, R. Vuillemot, and H. Pfister. "UpSet: Visualization of Intersecting Sets". In: *IEEE Transactions on Visualization and Computer Graphics* 20.12 (Dec. 2014), pp. 1983–1992.

[Lo+09] David Lo, Hong Cheng, Jiawei Han, Siau-Cheng Khoo, and Chengnian Sun. "Classification of Software Behaviors for Failure Detection: A Discriminative Pattern Mining Approach". In: *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2009, pp. 557–566. DOI: 10.1145/1557019.1557083.

[LS18] Caroline Lemieux and Koushik Sen. "FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage". In: *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering*. 2018, pp. 475–485.

[LT93] Nancy G. Leveson and Clark S. Turner. "An Investigation of the Therac-25 Accidents". In: *Computer* 26.7 (July 1993), pp. 18–41. DOI: 10.1109/MC.1993.274940.

[Lu+14] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. "A Study of Linux File System Evolution". In: *ACM Transactions on Storage* 10.1 (Jan. 2014), 3:1–3:32. DOI: 10.1145/2560012.

[Luk+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2005, pp. 190–200. DOI: 10.1145/1065010.1065034.

[Lyn18] Gerald Lynch. *Amazon Key smart lock security integrity called into question by hack*. Feb. 2018. URL: https://www.techradar.com/news/amazon-key-smart-lock-security-integrity-called-into-question-by-hack (visited on 03/05/2020).

[LZE15] Wing Lam, Sai Zhang, and Michael D. Ernst. *When tests collide: Evaluating and coping with the impact of test dependence*. Tech. rep. University of Washington Department of Computer Science and Engineering, 2015.

[Mah+12] R. Mahmood, N. Esfahani, T. Kacem, N. Mirzaei, S. Malek, and A. Stavrou. "A whitebox approach for automated security testing of Android applications on the cloud". In: *Proceedings of the 7th International Workshop on Automation of Software Test*. 2012, pp. 22–28. DOI: 10.5555/2663608.2663613.

[Mat16] Lucas Matney. *Nest's Smart Home Apps Are Back Online Following Outages*. Jan. 2016. URL: https://techcrunch.com/2016/01/09/nests-smart-home-apps-are-back-online-following-outages (visited on 03/05/2020).

[Mis+07] Sasa Misailovic, Aleksandar Milicevic, Nemanja Petrovic, Sarfraz Khurshid, and Darko Marinov. "Parallel Test Generation and Execution with Korat". In: *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ESEC-FSE '07. New York, NY, USA: ACM, 2007, pp. 135–144. DOI: 10.1145/1287624.1287645.

[MITa] MITRE. *CVE-2014-7186*. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-7186 (visited on 03/06/2020).

[MITb] MITRE. *CVE-2015-1538*. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1538 (visited on 03/18/2020).

[MITc] MITRE. *CVE-2015-6602*. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-6602 (visited on 03/18/2020).

[Nat+13] R. Natella, D. Cotroneo, J.A. Durães, and H.S. Madeira. "On Fault Representativeness of Software Fault Injection". In: *IEEE Transactions on Software Engineering* 39.1 (Jan. 2013), pp. 80–96. DOI: 10.1109/TSE.2011.124.

[Nat11] Roberto Natella. "Achieving Representative Faultloads in Software Fault Injection". PhD thesis. Università di Napoli Federico II, 2011.

[Nat13] Roberto Natella. *SAFE: SoftwAre Fault Emulator tool*. 2013. URL: http://wpage.unina.it/roberto.natella/tools.html.

[NCM16] Roberto Natella, Domenico Cotroneo, and Henrique S. Madeira. "Assessing Dependability with Software Fault Injection: A Survey". In: *ACM Computing Surveys* 48.3 (Feb. 2016), 44:1–44:55. DOI: 10.1145/2841425.

[NS07a] Nicholas Nethercote and Julian Seward. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2007, pp. 89–100. DOI: 10.1145/1250734.1250746.

[NS07b] Nicholas Nethercote and Julian Seward. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2007, pp. 89–100. DOI: 10.1145/1250734.1250746.

[Orm] Tavis Ormandy. *Cloudbleed*.

[OU10] M. Oriol and F. Ullah. "YETI on the Cloud". In: *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*. Apr. 2010, pp. 434–437. DOI: 10.1109/ICSTW.2010.68.

[Pad+08] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. "Documenting and Automating Collateral Evolutions in Linux Device Drivers". In: *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*. Eurosys '08. Glasgow, Scotland UK: ACM, 2008, pp. 247–260. DOI: 10.1145/1352592.1352618.

[Pal+11] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. "Faults in Linux: Ten Years Later". In: *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVI. ACM, 2011, pp. 305–318. DOI: 10.1145/1950365.1950401.

[Par+09] T. Parveen, S. Tilley, N. Daley, and P. Morales. "Towards a distributed execution framework for JUnit test cases". In: *2009 IEEE International Conference on Software Maintenance*. Sept. 2009, pp. 425–428. DOI: 10.1109/ICSM.2009.5306292.

[Pet+17] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. "SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities". In: *2017 ACM Conference on Computer and Communications Security*. 2017, pp. 2155–2168.

[Phi] Philips. *Philips Hue Bridge release notes*. URL: https://www2.meethue.com/en-us/support/release-notes/bridge (visited on 03/05/2020).

[Pip+12] T. Piper, S. Winter, P. Manns, and N. Suri. "Instrumenting AUTOSAR for dependability assessment: A guidance framework". In: *2012 International Conference on Dependable Systems and Networks*. 2012, pp. 1–12. DOI: 10.1109/DSN.2012.6263913.

[Pip+15] T. Piper, S. Winter, N. Suri, and T. E. Fuhrman. "On the Effective Use of Fault Injection for the Assessment of AUTOSAR Safety Mechanisms". In: *Proceedings of the 11th European Dependable Computing Conference*. 2015, pp. 85–96. DOI: 10.1109/EDCC.2015.14.

[Pri09] Princeton University. *The PARSEC Benchmark Suite*. 2009. URL: http://parsec.cs.princeton.edu/parsec3-doc.htm.

[Raw+17] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. "VUzzer: Application-aware evolutionary fuzzing". In: *Proceedings of the Network and Distributed System Security Symposium*. 2017.

[Reb+14] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. "Optimizing Seed Selection for Fuzzing." In: *Proceedings of USENIX Security*. 2014, pp. 861–875.

[Reg05] John Regehr. "Random Testing of Interrupt-driven Software". In: *Proceedings of the 5th ACM International Conference On Embedded Software*. 2005, pp. 290–298. DOI: 10.1145/1086228.1086282.

[Rod+99] Manuel Rodríguez, Frédéric Salles, Jean-Charles Fabre, and Jean Arlat. "MAFALDA: Microkernel Assessment by Fault Injection and Design Aid". In: *Proceedings of the Third European Dependable Computing Conference*. Ed. by Jan Hlavička, Erik Maehle, and András Pataricza. 1999, pp. 143–160.

[SAM08] Akbar Siami Namin, James H. Andrews, and Duncan J. Murdoch. "Sufficient Mutation Operators for Measuring Test Effectiveness". In: *Proceedings of the 30th International Conference on Software Engineering*. ICSE'08. 2008, pp. 351–360. DOI: 10.1145/1368088.1368136.

[Sch+15] H. Schirmeier, M. Hoffmann, C. Dietrich, M. Lenz, D. Lohmann, and O. Spinczyk. "FAIL*: An Open and Versatile Fault-Injection Framework for the Assessment of Software-Implemented Hardware Fault Tolerance". In: *Proceedings of the 11th European Dependable Computing Conference*. 2015, pp. 245–255. DOI: 10.1109/EDCC.2015.28.

[Sch+17] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. "kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels". In: *Proceedings of USENIX Security*. 2017, pp. 167–182.

[Sch+18a] Oliver Schwahn, Nicolas Coppik, Stefan Winter, and Neeraj Suri. "FastFI: Accelerating Software Fault Injections". In: *23rd IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*. PRDC 2018. Taipei, Taiwan, Dec. 2018, pp. 193–202. DOI: 10.1109/PRDC.2018.00035.

[Sch+18b] Oliver Schwahn, Stefan Winter, Nicolas Coppik, and Neeraj Suri. "How to Fillet a Penguin: Runtime Data Driven Partitioning of Linux Code". In: *IEEE Transactions on Dependable and Secure Computing* 15.6 (Nov. 2018), pp. 945–958. DOI: 10.1109/TDSC.2017.2745574.

[Sch+19] Oliver Schwahn, Nicolas Coppik, Stefan Winter, and Neeraj Suri. "Assessing the State and Improving the Art of Parallel Testing for C". In: *28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2019. Beijing, China: ACM, 2019, pp. 123–133. DOI: 10.1145/3293882.3330573.

[Ser+12] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. "AddressSanitizer: A Fast Address Sanity Checker". In: *Proceedings of the 2012 USENIX Annual Technical Conference*. 2012, pp. 309–318.

[SP10] Matt Staats and Corina Păsăreanu. "Parallel Symbolic Execution for Structural Test Generation". In: *Proceedings of the 19th International Symposium on Software Testing and Analysis*. ISSTA '10. 2010, pp. 183–194. DOI: 10.1145/1831708.1831732.

[SPS09] M. Sand, S. Potyra, and V. Sieh. "Deterministic high-speed simulation of complex systems including fault-injection". In: *2009 International Conference on Dependable Systems and Networks*. 2009, pp. 211–216. DOI: 10.1109/DSN.2009.5270335.

[SS75] J. H. Saltzer and M. D. Schroeder. "The protection of information in computer systems". In: *Proceedings of the IEEE* 63.9 (Sept. 1975), pp. 1278–1308. DOI: 10.1109/PROC.1975.9939.

[Sta00] E. Starkloff. "Designing a parallel, distributed test system". In: *Proceedings of IEEE AUTOTESTCON 2000*. 2000, pp. 564–567.

[Syn] Synopsys. *Heartbleed*. URL: http://heartbleed.com/ (visited on 03/06/2020).

[Sys] SystemTap. *SystemTap*. URL: https://sourceware.org/systemtap/ (visited on 03/06/2020).

[TKL18] Rijnard van Tonder, John Kotheimer, and Claire Le Goues. "Semantic Crash Bucketing". In: *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering*. 2018, pp. 612–622.

[TP13] Anna Thomas and Karthik Pattabiraman. "LLFI: An intermediate code level fault injector for soft computing applications". In: *Proceedings of the IEEE Workshop on Silicon Errors in Logic - System Effects*. 2013.

[Voa+97] J. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman. "Predicting How Badly "Good" Software Can Behave". In: *IEEE Software* 14.4 (1997), pp. 73–83.

[Vyu] Dmitry Vyukov. *syzkaller*. URL: https://github.com/google/syzkaller (visited on 03/06/2020).

[Wan+17] J. Wang, B. Chen, L. Wei, and Y. Liu. "Skyfire: Data-Driven Seed Generation for Fuzzing". In: *IEEE Security & Privacy*. 2017, pp. 579–594.

[Win+13] Stefan Winter, Michael Tretter, Benjamin Sattler, and Neeraj Suri. "simFI: From Single to Simultaneous Software Fault Injections". In: *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, June 2013, pp. 1–12. DOI: 10.1109/DSN.2013.6575310.

[Win+15] Stefan Winter, Oliver Schwahn, Roberto Natella, Neeraj Suri, and Domenico Cotroneo. "No PAIN, No Gain?: The Utility of PArallel Fault INjections". In: *Proceedings of the 37th International Conference on Software Engineering*. ICSE '15. Florence, Italy: IEEE Press, 2015, pp. 494–505. DOI: 10.1109/ICSE.2015.67.

[Won+16] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. "A Survey on Software Fault Localization". In: *IEEE Transactions on Software Engineering* 42.8 (Aug. 2016), pp. 707–740. DOI: 10.1109/TSE.2016.2521368.

[Yu+09] Lian Yu, Le Zhang, Huiru Xiang, Yu Su, Wei Zhao, and Jun Zhu. "A Framework of Testing as a Service". In: *2009 International Conference on Management and Service Science*. Sept. 2009, pp. 1–4. DOI: 10.1109/ICMSS.2009.5302717.

[Yu+10] Lian Yu, Wei-Tek Tsai, Xiangji Chen, Linqing Liu, Yan Zhao, Liangjie Tang, and Wei Zhao. "Testing as a Service over Cloud". In: *2010 Fifth IEEE International Symposium on Service Oriented System Engineering*. 2010, pp. 181–188. DOI: 10.1109/SOSE.2010.36.

[Zal] Michal Zalewski. *American Fuzzy Lop*. URL: http://lcamtuf.coredump.cx/afl/ (visited on 03/06/2020).

[ZE13] Sai Zhang and Michael D. Ernst. "Automated Diagnosis of Software Configuration Errors". In: *Proceedings of the 35th International Conference on Software Engineering*. 2013, pp. 312–321.

[Zel02] Andreas Zeller. "Isolating Cause-effect Chains from Computer Programs". In: *Proceedings of the Tenth ACM SIGSOFT Symposium on Foundations of Software Engineering*. 2002, pp. 1–10.

[Zha+14] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D. Ernst, and David Notkin. "Empirically Revisiting the Test Independence Assumption". In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ISSTA'14. New York, New York, USA: ACM Press, 2014, pp. 385–396. DOI: 10.1145/2610384.2610404.

[Zha+18] G. Zhang, X. Zhou, Y. Luo, X. Wu, and E. Min. "PTfuzz: Guided Fuzzing With Processor Trace Feedback". In: *IEEE Access* 6 (2018).

[ZKB13] Bowen Zhou, Milind Kulkarni, and Saurabh Bagchi. "WuKong: Effective Diagnosis of Bugs at Large System Scales". In: *SIGPLAN Notices* 48.8 (Feb. 2013), pp. 317–318. DOI: 10.1145/2517327.2442563.